COMPARISON OF NUMERICAL RESULT CHECKING MECHANISMS FOR FFT COMPUTATIONS UNDER FAULTS

A Thesis
Presented to
The Academic Faculty

by

Saraswati Bharthipudi

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2003

COMPARISON OF NUMERICAL RESULT CHECKING MECHANISMS FOR FFT COMPUTATIONS UNDER FAULTS

Approved by:

Dr.Douglas Blough, Committee Chair

Dr.David Schimmel, Adviser

Dr.Feodor Vainstein

12/08/2003

Date Approved

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my advisor Dr.David Schimmel who guided me through this endeavor. I would also like express special thanks to Dr.Doug Blough for his valuable suggestions and guidance. I would also like to thank Dr.Vainstein whose algorithm has been studied and implemented in this thesis. Finally I would like to thank my husband Prashant for his support and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS

τ : The difference obtained in the post-condition check between the LHS and RHS (described in chapter 6)

δ : The difference obtained in the post-condition check between the LHS and RHS divided by a scaling (described in chapter 6)

EDM 1 : FFT checking algorithm proposed by Dr. Feodor Vainstein

EDM 2 : FFT checking algorithm proposed in paper [1]

EDM 3 : FFT checking algorithm based on Parseval's test

SUMMARY

This thesis studies and compares existing Numerical Result checking algorithms for FFT computations under faults. In order to simulate faulty conditions, a fault injection tool is implemented. The fault injection tool is designed so as to be as non-intrusive to the application as possible. Faults are injected into memory in the form of bit flips in the data elements of the application. The performance of the three result checking algorithms under these conditions is studied and compared. Faults are injected at all the stages of the FFT computation by flipping each of the 64-bits in the double-precision representation. Experiments also include introducing random bit flips in the data array, emulating a more real-life like scenario. Finally the performance of these algorithms under a set of worst-case inputs is also studied.

CHAPTER I

INTRODUCTION

In my thesis I study and compare the performance of existing Numerical Result Checking algorithms for FFT under faulty conditions. The study also involves developing a software fault injection mechanism that simulates faulty environments.

Numerical Result checking is a popular algorithm based error detection mechanism for applications performing numerical computations. It is common for applications which are prone to faulty conditions like bit flips in memory to resort to error detection mechanisms that detect these occurrences. In numerical result checking, the output or the result of the application is checked to detect any errors introduced in the application due to faults in memory.

The result checker relies on post-conditions[1] that the computed output must satisfy in order validate the accuracy of the computation. This check for the post-conditions has to be done in a more numerically efficient manner than having to re-compute the output.

The result checker must be able to distinguish round-off errors from errors introduced due to faults. If the difference between the computed output and expected output lies within a threshold value, it is considered a round-off error. Any error larger than this threshold is considered as an error due to a fault. Choice of this threshold value is an important parameter and is discussed in detail in Chapter 7.

This thesis focuses on Numerical result checking algorithms that exist for applications

computing the Discrete Fourier Transform of input vectors. Discrete Fourier Transforms are used extensively in RADAR applications[26], which are susceptible to such fault conditions that can lead to errors in the computed output. This phenomenon is discussed in detail in the next chapter. While our study is focused on the DFT, each of the error detection approaches being compared can be generalized to all unitary transformations such as discrete cosine transformation (DCT). A unitary transformation is one in which the transform matrix is unitary, i.e. $A^{-1} = A^{*T}$.

The study involves evaluating existing Numerical result checking algorithms for DFT and comparing them under various parameters that are important in the performance of an error detection mechanism. Each of these parameters is subsequently introduced in this document. The Fault Injection tool implemented in order to inject faults into the application is also described.

Experiments involve subjecting the FFT application under study to faults at various instances and locations and evaluating the performance of each of the error-detection mechanism being studied. Experiments are designed so as to evaluate the accuracy, efficiency and coverage of each of these EDMs and a subsequent set of runs attempts to evaluate their performance under various real-life-like faulty scenarios by introducing a pseudo-randomness to the injected faults. A final set of runs involves a set of worst-case inputs generated with sufficiently high zero-values in them, and the performance of each of these EDMs is studied.

CHAPTER II

BACKGROUND AND RELATED WORK

The study involves evaluating the performance of error detection mechanisms under memory faults. There exist a number of conditions that can lead to such faults. Our primary interest is transient faults that may be caused by single-event-upsets (SEUs). In SEUs, high-energy particles produce spurious electrical signals, causing the state of transistors in a memory cell to reverse, effectively causing the memory cell to change state from 1 to 0, or 0 to 1. This phenomenon is especially common in space-borne computers where memory is exposed to such high-energy radiation. Memory faults are also caused by leakage of charge in transistors, causing the value stored in a memory cell to change from 1 to 0. Gate-level faults can result in the memory cell being stuck at 0 or 1. Other types of memory faults are faulty address decoders, faults in the address/data line etc. As mentioned earlier, we will be injecting memory faults that are transient in nature. Memory faults can be permanent or transient in nature. Permanent faults can be emulated by injecting repeated memory faults at a very high frequency.

Faults in memory could be latent or could manifest into errors. Latent memory faults are faults that occur at a location in memory that never gets accessed or gets subsequently overwritten.

2.1 Round-off errors

Round-off errors arise because of the representation of real numbers with finite precision in Computers (Please see section 7.4 for IEEE representation of double precision numbers). In general round-off error growth is linear in the number of operations. Although roundoff error in a single numerical operation is small , there are situations where the roundoff error can become significant, or even catastrophic. For example when subtracting two numbers of nearly equal value, and one or both of them have incurred round-off errors already, then the difference between the two will result in a very small number and hence a very large relative error. If x is the actual number and $\hat{x}$ is its truncated representation , then *Relative error* is defined as $|(x - \hat{x})/x|$. *Absolute error*is defined as $|x - \hat{x}|$. Dividing a small number by a very large number can result in a large relative error. The accumulation of round-off errors depends on the way a computation in implemented. For example when a small number is subtracted from a large number and this difference is again subtracted from a large number we have a large round-off error. Instead if the computation is implemented such that the two large numbers are subtracted first to obtain a small number which is then subtracted from the small number, we avoid the large round off error.

2.2 Error Detection Mechanisms

Software Error Detection Mechanisms can be broadly classified into in-built error detection at the operating system level and error detection at the application level.

Operating systems have in-built error-detecting mechanisms that signal an error to the application under certain faulty conditions - for example when a malformed instruction is encountered (illegal instruction), or when the application tries to access an address that

does not exist or it does not have access to (segmentation violation), etc. Operating system alerts the application of such errors via directed signals. Work has been done in evaluating error-detection capabilities at an operating system level. Steininger et al. have studied the effectiveness of various error-detection mechanisms in a Self-Checking RISC board by injecting pin-level hardware faults[3]. Our study deals with the error-handling capability of an application, built over the error-detecting capability of the operating system.

The virtual address space of an application process is divided into text segment that stores instructions, static data segment that stores global variables and dynamic data segments including heap and stack that store dynamically allocated data and dynamically declared variables respectively.

Faults could occur at different segments of the memory address space. Faults in the Text segment are relatively easy to detect. An illegal instruction or a Bus error is easily caught by the Operating System and signaled to the application.

Data errors, i.e. errors in the static or dynamic data segments are more difficult to detect, unless the address of a variable is modified to an illegal address or to an address that exceeds the allocated segment boundaries. Thus any bit flip that mutates data to another legal data is not detected by the Operating System. Therefore applications that reside in an environment that is prone to such faults in memory (described earlier) resort to additional fault tolerance techniques or error detection mechanisms. Popular fault tolerance techniques include introducing hardware and software redundancies like n-version programming and voting mechanisms. However redundancies add to overhead.

Application specific error detection mechanisms check the logical correctness of execution state based on application specific knowledge. Numerical Result checking is a

popular algorithm based error detection mechanism, which can be applied to applications performing numerical computations. This approach has the distinct advantage of being more efficient than the original algorithm since it relies on post condition checks rather than re-computing the result.

Wasserman and Blum [2] talk about the need for a Result checker to be time-bound in Real-time systems, i.e the checker has to complete execution with a bounded time T(n). While several fault tolerance mechanisms have been proposed for FFT[2-8], they study gate level faults and concurrent error detection schemes. Not only is this at a lower abstraction than our interest, it requires special additional hardware for its implementation. This is not a viable solution when the industry trend is towards COTS (commercial off-the-shelf) components. Moreover they are specific to a fast algorithm while the error detection mechanisms under our study can be applied to any Fourier transformation independent of the underlying fast algorithm.

Redinbo [9] suggests a generic numeric expression for Unitary transformations that could be used to detect errors in FFT. They suggest error detection based on comparing two parity values, one computed by the weighted sum of the transform coefficients and the other from a weighted sum of the input data. While this is an approach similar to the EDMs we are evaluating, an implementation and detailed analysis of its coverage, performance etc. has not been done yet. [1] has implemented their algorithm and measured its coverage by randomly injected bit flips at random stages of the FFT computation. However a detailed study of the performance of this algorithm under varying vector sizes and for faults injected at each bit and at each stage of the FFT computation has not been done.

It is interesting to study the behavior of faults at each stage of the FFT algorithm as well

as the coverage of each error detection mechanism under such faults. This is because a fault at a given bit at stage x might have a distinct effect from a fault at the same bit at stage x+k. While theoretical expressions for the error propagation through the various stages in FFT have been derived [27], an implementation and a practical study of this phenomenon and its effect on various EDMs has not been done.

Finally we propose to study the behavior of these EDMs under a set of worst-case inputs containing members that are zero or near zero with a few elements that are of extremely large magnitudes leading to catastrophic-cancellations. The behavior of each of the EDMS when subjected to such adverse inputs is studied. [22] introduced the concept of Algorithm Based Fault Tolerance using check-sums for systolic arrays. [23] compares Algorithm based fault tolerance with Result checking for Matrix operations like Matrix multiplication, QR decomposition, Matrix Inversion,etc. [24] suggests the idea of stored-randomness and pre-computing in order to reduce the run-time overhead. [25] discusses the advantages of inserting run-time result checkers in embedded Information systems. [28] introduces algorithm-based fault tolerance for FFT processors, but again requires additional hardware for its implementation.

Various fault injection mechanisms have been proposed, where faults are either injected into the hardware or faults are simulated using software mechanisms[12]. Earlier fault-injection mechanisms were primarily hardware fault injections where additional hardware was used to inject faults. MESSALINE, is one such general pin-level fault injection tool[13]. FIAT injects transient faults into a chip using heavy ion radiation[14]. However, in recent years software fault injection techniques have been gaining popularity. This is because they provide more precise control over location and timing of faults and require no

additional hardware. Software mechanisms emulate hardware memory or communication faults using software. We shall use software mechanisms to inject all faults because they offer better control from an application-level point of view. The evolution of software fault injection mechanisms is discussed in detail in the following section.

2.3 Software Fault Injection Mechanisms

2.3.1 Fault Injection approaches

Faults can be injected into an application pre-run time, where the application is modified before it is loaded or at run-time where a fault injection mechanism injects faults, while the application is executing. Both approaches have their advantages and disadvantages. In pre-run time fault injection, the source code or assembly-level or machine-level code is modified. Thus pre-run time fault-injection, requires the application to be re-compiled and/or re-loaded every time fault parameters are changed and also requires modification of the application code. Another disadvantage of this approach is that faults cannot be specified while the application is executing. In run-time fault injection, it is possible to inject faults at certain time offsets of application execution, thus giving the user control of the precise instant of fault injection. However, this fault injection mechanism can be intrusive to the application execution, affecting the application's run-time behavior.

2.3.2 Memory model

FERRARI[15] emulates faults in data line, address line, CPU and memory - a fairly low level of abstraction. Effects of faults in the CPU including faults in data registers, address registers, ALU, etc are highly dependent on the processor architecture.

Ours is a flattened view of memory as opposed to the hierarchical memory model consisting of cache, main memory, etc. This model is architecture-independent. We have adopted the address space memory model in our study, because we are primarily interested in fault injection at an application-level perspective. Also, faults in the virtual address space indirectly emulate the above mentioned low-level faults, for example, a faulty address could be because of a faulty register, or faulty memory cell or faulty address/data line in the bus. Note that in FERRARI, faults are injected at run-time by trapping program execution at specified instructions or by timers(This approach has the disadvantage of being intrusive to program execution.)[15]. Faults are injected by modifying instruction or memory

2.3.3 Related work in fault injection

DOCTOR, has a similar approach as ours, injecting faults into the process address space but on a real-time distributed environment called HARTS[16]. However, like most real-time systems, HARTS does not employ memory protection. Thus additional challenges in our mechanism include modifying protected memory segments like the text segment and dealing with the possibility of the fault injector causing page faults and thereby altering the natural flow of execution of the program[1]. In DOCTOR, communication faults are injected by intercepting send/receive operations and changing message contents. In contrast, our tool supports the distributed object environment of CORBA, where communication is essentially through remote object calls with input, output and inout parameters. Fault-injection is therefore done at a much higher-level of abstraction than in DOCTOR. FIAT is another such real-time fault injection tool injecting faults into memory as well messages[13]. In

[1]Real time systems pre-allocate memory in order to avoid page faults

DOCTOR communication faults are injected by intercepting send/receive operations and changing message contents[16]. Work has been done on fault injection at the Operating-system and middle-ware level. FINE is a fault injection mechanism that injects faults into the UNIX operating system (as opposed to applications, which is our area of study) and observes the system behavior[17]. Similarly, Chung et.al[18] have studied the fault injection and behavior of popular CORBA and DCOM implementations on the NT platform while Orchestra[19] corrupts messages to test dependability of communication protocols. FTAPE injects faults into memory, the target here is the processor[18]. A synthetic workload generator creates high-stress conditions for the processor and the performance of the processor in the faulty environment is studied. GOOFI injects pin-level and pre-run time faults(i.e. faults are injected before the application starts to execute)into memory[21].

Some and Kim et al. have injected single bit memory faults into an FFT application and found that heap is most significant for their application, as it occupied a majority of the application memory[20]. ESFFI injects faults into COTS(Commercial off-the-shelf) components, by trapping specified instructions and injecting faults(disadvantage of being intrusive)[21].

The objective of our fault-injection tool is to support the distributed object environment of CORBA on a unix-based platform, injecting both memory and communication faults into applications at run-time.The mechanism is as non-intrusive to application execution as possible and allows the user to specify faults pre-run time(but post-compile time)as well as at run time(via API calls). The tool allows the user a wide range of choices on fault location, model and timing. Our fault-injection mechanism is described in detail in the following section.

CHAPTER III

FAULT INJECTOR

In this chapter we discuss the features of our fault injector and various types and conditions of faults that can be specified for injection. We also describe the strategy adopted to make it as non-intrusive as possible.

The fault injection mechanism by our tool is primarily run-time fault injection, where the user can specify faults both pre- and post-compile time. The tool henceforth referred to as FIMA (Fault Injector for Middleware Applications) is capable of injecting memory and communication faults for distributed applications. The fault injector is a thread that attaches itself to the main application process, wakes up at the specified time of fault injection and injects faults into memory, does some fault-queue management and mostly sleeps otherwise. This considerably reduces intrusion into the run-time behavior of the application. The fault injector is an event driven entity where events constitute specified faults to be injected, remote method invocations and any communication from the main application via API calls. For every remote method invocation, the fault injector has to wake up and update its untimed queues that store faults based on method invocation count.

The event driven mechanism is realized by utilizing system timers in the real-time library. Solaris offers both wall-clock as well as process-execution time based timers. A timer expiration results in the receipt of the signal SIGALRM and the timer id of the timer that has just expired. The signal handler determines if the timerid is one of FIMA fault

timers. If yes, it wakes up the FIMA thread.

The user also has the option of specifying faults to be temporally dependent on instruction execution. This is made available to the user via API (Application programmer interface) calls, and requires source modification and application compilation. Thus the fault actually gets specified at run-time and can have run-time dependencies, such as inject fault on the i^{th} invocation of a method, or inject fault if a condition is true. This allows for a very precise control of the location of faults like data variables etc, which is useful for our experiments.

Memory fault injection is done by flipping, setting or resetting bits at the specified location in the address space i.e text space, static data space, heap or stack. In order to make sure the fault injection thread does not cause page faults, thereby altering application run-time behavior, the fault injector checks to make sure the page containing fault location is in fact existing in main-memory. If not, the fault is not injected, and this is appropriately logged.

Also, since the fault-injector is a thread and shares its address space with the application process, we must restrain the fault-injector from corrupting its own code or data space. This is addressed in the following way. The fault Injector code resides as a library that needs to be linked in with the application object file. By compiling the library as a shared library that gets linked only at run-time, we ensure that the Fault Injector code and data resides in the section of the address space that is allocated for shared libraries: between the upper limit of the heap and the lower limit of the stack. Since our Fault Injection tool constantly checks the dynamic data limits to ensure faults are *not* injected in this region, we thus restrain the fault injector from corrupting itself.

3.1 Run time overhead

In a multi-processor machine the fault injector could be bound to a light-weight process which would totally eliminate any run-time overhead on the main application. In order to determine the run-time overhead of the fault injector on the application, we consider its execution on a single processor machine. A cpu-intensive application is a good application example for evaluating this, since it competes with the fault injector for CPU resources. This is true for all threads that are unbounded, i.e. they have a process contention scope. This means they compete with other threads within the process for getting scheduled. While having the thread bound to a light-weight process eliminates its contention with the main application for getting scheduled, a thread switch now also involves a light-weight process switch in a single-processor machine. This is obviously more expensive than a simple thread switch in the unbound thread case. We therefore do not bind our fault injector to any light-weight process.

It is also important to understand the scheduling policies of the operating system. Time-sharing (TS), which is the default scheduling policy in Solaris, is not a real-time scheduling policy and the scheduler dynamically changes the priority of processes/threads based on the process activity. A process that sleeps or yields quickly is considered to be highly interactive and the scheduler therefore increases its priority. A cpu-bound process/thread is pushed down in priority. There is no bounded behavior for such a scheduling policy. In real-time scheduling, we have FIFO and Round-Robin where there is bounded time within which a thread will be scheduled. The scheduler does not dynamically alter the process priority. However setting the scheduler to real-time would require root access since

real-time scheduling priority is considered higher than time-sharing (which is the default scheduling policy of the process). The fault injector thread would have to relinquish its allocated cpu-time through an explicit call to yield during its idle time.

In making the thread event driven, we instead determine when it will be scheduled and avoid being scheduled during idle time and making explicit calls to sched-yield, involving unnecessary thread-switches. The fault injector thread essentially blocks on a mutex condition variable during idle time. Before this, it sets the system timers to fault injection time of the fault in the queues' head (Fault queues are sorted based on fault injection time). Any event like the expiry of a timer, an API call from the main application or a remote method invocation results in a call to signal the condition variable and the system wakes up the fault injector thread. This mechanism thus involves considerable thread synchronization using mutexes and avoiding race conditions.

In order to determine the run-time overhead of the fault injector, we consider its execution on a single processor machine. We choose the FF result checking application which is a CPU-intensive application.

We baseline the application execution time without the fault injector attached to it. We then attach the Fault injector but run it with no faults specified and measure the execution overhead. Finally we run a set of experiments with single and multiple faults injected. Multiple faults are injected at the rate of one every 0.04 secs which results in around 3-4 faults being injected per run.

Note that these set of readings exclude the initialization time required by the FIMA thread to initialize its data structures. CPU times shown below represent the average execution time from 100 runs. CPU time represents the sum of CPU times of the main

application and the fault injector thread. We observe that execution time of the application with FIMA thread under no faults is almost equal to its execution time without the FIMA thread. This is because in the absence of faults, the FIMA thread essentially sleeps.

In the above described experiments, faults have been specified interms of offsets from the address space. For such faults, the fault injector determines the dynamic address space limits before injecting the specified fault. This is not done for faults specified as offsets from static address segments. Thus faults in these experiments represent a worst-case per fault overhead.

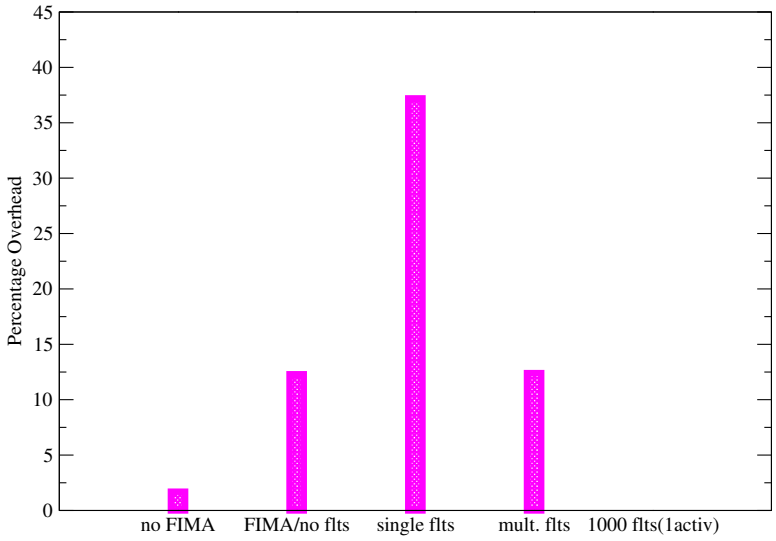

Figure 1: FIMA execution overhead: Compares execution times of application in the absence of FIMA thread, with FIMA thread but no faults, Single fault injected per run, multiple faults at faults every 0.04 secs. and with a 1000 faults specified in future and only one fault being active(i.e only 1 fault gets injected)

The last bar in the graph represents a 1000 faults specified in the Fault specification file, out of which only one is active. Note that the 1000 static faults have not caused an overhead on the execution. This is because the fault queues are time sorted and the FIMA thread only checks the head of the queue, rather than parsing through the entire queue.

We now evaluate the initialization overhead of the fault injector thread. For this again, we measure the cpu-time for the FIMA thread to initialize and return. This overhead is independent of the application execution time. The only variable component in this overhead could be the size of the fault specification file. Thus we have below the average cpu-overhead from 100 runs, when run with no faults specified and the size of fault specification file is the minimum possible with 200 memory and communication faults and finally with a 1000 memory and communication faults. The average per fault overhead at startup is approximately 16 micro-seconds.

We see that FIMA has a minimum initialization overhead of 0.0636. The API calls that specify a fault were found to take around 0.322 milli-secs per fault (averaged over a hundred runs) in the absence of static faults. With a 1000 faults specified in the queues (500 memory faults with the same time unit as the API calls), the average time for the API calls was 0.339 mill-seconds per fault. This is important, because the API call involves sorting through the queue to find the appropriate position to insert. Our API calls were such that their start time was greater then the start times of all the statically specified faults. This means, the API call has to search through the entire queue before inserting the fault each time.

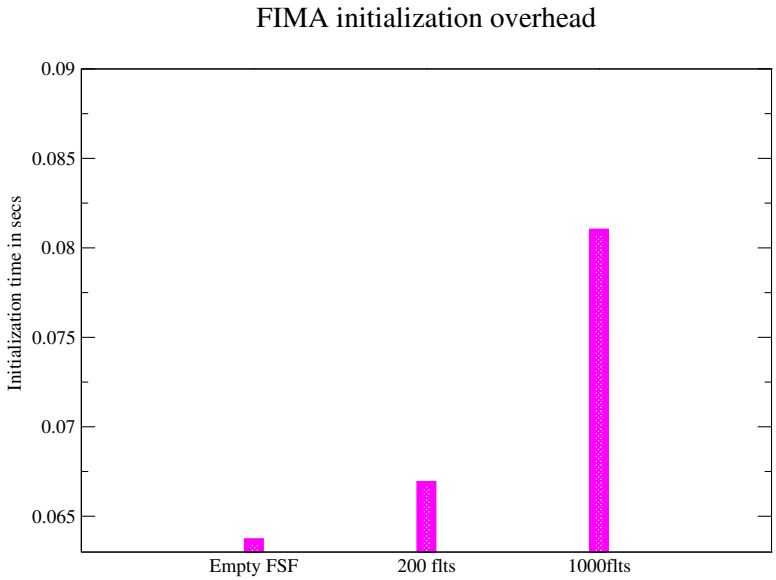

Figure 2: FIMA initialization overhead: With an empty Fault spcecification file, 200 static faults and 1000 static faults

3.2 Memory limits

The fault injector has to inject faults only within allocated memory limits to avoid causing page faults thereby altering the natural execution flow of the main application. In order to do this it is necessary to determine the address limits of the various segments of the address space. Static limits like the text segment and static data segment can be obtained from the executable. GNU utility objdump parses the executable and determines the static address limits.

Dynamic limits are obtained at run time using the /proc filesystem in Solaris. The /proc file system stores a copy of the virtual address space of each executing process. Dynamic memory limits like the heap and stack limits are obtained at run-time by reading from this filesystem.

If the user specifies a memory location for fault injection that is outside these limits, an

error is logged and the fault is not injected.

CHAPTER IV

EFFECTS OF FAULTS

In this chapter we present the possible effects of faults in the various segments in the address space. Faults in each of the address segments can result in different behaviors of the same application. Faults can lead to the application crashing, can lead to errors in the outputs of the application or can be latent and not affect the execution of the application at all.

4.1 Faults in the text segment

The various possible effects of faults in the text segment can be summarized as follows:

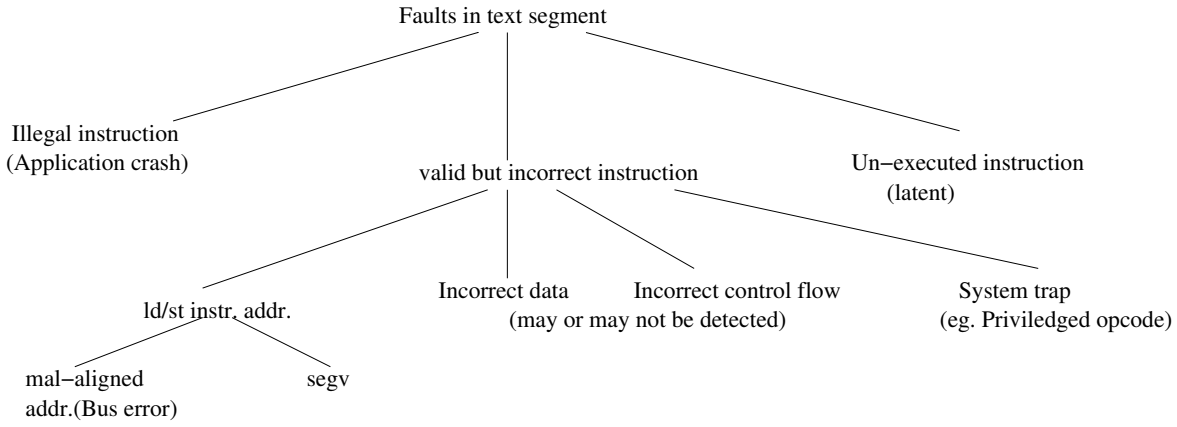

Figure 3: Faults in text segment

Faults in the code segment can lead to the execution of an illegal or malformed instruction, causing the operating system to raise an exception of Illegal Instruction causing the application to crash. The fault could mutate the code in such a way that the modified instruction is a legal instruction. This could result in the instruction executing and giving

incorrect results, thereby corrupting data or the mutated instruction could cause an incorrect branching operation, altering the application control flow. If the mutated instruction is now a privileged opcode, it will result in the operating system raising an exception and the application crashing. Faults in the text segment could also modify a load/store instruction and the address offsets in the operands get modified in such a way, that the application tries to access an un-aligned address, causing the system to throw an exception of Bus-error or an illegal address is accessed causing the system to throw an exception of Segment Violation. If the address gets modified to a valid but incorrect address, then this could result in incorrect data flow and/or incorrect results. The fault could also be latent in nature, where the modified instruction never gets executed.

Thus we see that in most cases, a fault in the text segment that manifests into an error is caught by the Operating System.

4.2 Faults in the data segment

The various possible effects of faults in the data segment are summarized below:

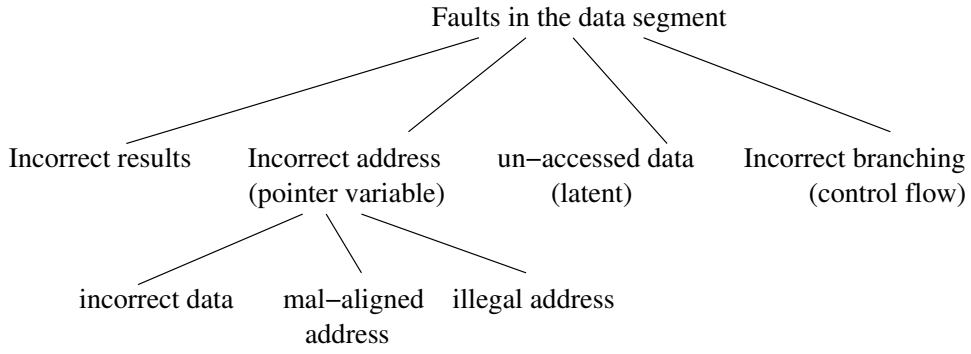

Figure 4: Faults in data segment

The immediate impact of a fault in the data segment could be incorrect results because of reading corrupted data. A data fault could also cause an incorrect branching and thereby modify control flow, if data is mutated such that it affects the result of a branching instruction. It is also possible that the corrupted data is never subsequently read or gets over-written and there by the fault remains latent. The fault could also corrupt an address value if the fault affects a pointer variable. When the contents of a pointer variable are corrupted, i.e. it now points to an incorrect address; the new address could be a valid address (and therefore causes the application to access wrong data) or it could be a mal-aligned address, causing the operating system generate a signal (Bus error) when the application tries to access the contents of the pointer variable. It is also possible that the modified contents of the pointer variable is an illegal address, i.e. an address the application does not have read/write permissions to or an address that is not mapped to the application process.

4.3 Heap and Stack faults

4.3.1 Stack faults

Faults in the stack have a high probability of subsequent reads and therefore more likely to affect program execution than any other types of faults. Stack faults can either cause data corruption or they can alter the program control flow by corrupting the stack trace. Thus faults in the stack could either behave as data faults or code faults that cause incorrect control flow or application crash (because of segmentation violation, bus error). It is expected that control-oriented applications, would have higher stack traces at any point of execution, as compared to I/O-intensive applications, and therefore a higher probability of error due to such faults.

4.3.1.1 Heap faults

Faults in the heap might possibly have a higher impact on data intensive applications because of their larger heap and stack size. On the other hand, larger heap size could mean a larger area for faults to be scattered and therefore lower impact, depending on how the heap is accessed by the program after fault injection.

Thus we see that data errors are most difficult to detect and most errors in the text segment are detected by the in-built error detection mechanisms of the Operating System. Applications that are prone to such faults therefore have to rely on additional error-detection mechanisms.

Thus we see that data errors are difficult to detect. Moreover the FFT computation application usually stores its input and output vectors as dynamically allocated data. Since our application is most prone to faults in the heap, our set of experiments involve injecting faults into the heap of the application.

CHAPTER V

NUMERICAL RESULT CHECKING APPROACHES

In this chapter we introduce the three numerical result checking algorithms and present their mathematical proofs.

The post-condition to be satisfied by any DFT computation is:

$$y \approx W^T x$$

where x is the $n * 1$ input vector; W is the $n * n$ DFT transform matrix; y is the $n * 1$ output vector

Taking into account the round-off errors due to computation, the post-condition check becomes:

$$(y - W^T x) \leq v$$

where v is some invariant that accounts for round-off errors introduced in the computation.

We observe that v is a n-length vector and any numerical result checking involves comparing of two scalar values. In order to reduce this expression to a scalar, a probe vector w is introduced[1].

$$w^T(y - W^T x) \leq \tau \tag{1}$$

where $\tau = wv$ We refer to τ as the threshold representing the difference in the post-condition check.

5.1 EDM 1

EDM 1 is a generic solution that checks for the computational correctness of a linear mapping $A : F^n \rightarrow F^m$, where F is a field($\Re$, $\mathcal{C}$, or any other infinite of finite field). Described below is its application to the specific problem of checking the correctness for FFT computations.

Let $x = (x_1, x_2, x_3....x_n), x\mathrm{i} \in \Re$ be input data.

Then $FFT(x) = y = (y_1, y_2, ...y_n), \mathrm{y_i} \in C^n$

Thus we have a mapping

$$FFT : R^n \rightarrow C^n$$

This mapping will be treated as the mapping A in the preceding discussion.

Now, according to the suggested method, we have to chose a linear mapping

$$B : C^n \rightarrow \Re$$

Since C^n as a linear space over $\Re$ is isomorphic to $\Re^{2n}$, a mapping B is specified by a vector $b = (b_1, ...b_{2n})$

for $y = (y_1, ..., y_n) = (\alpha_1 + \jmath\beta_1, ...\alpha_n + \jmath\beta_n) \in C^n$ as:

$$B(y) = b_1\alpha_1 + b2\beta_1 + b_3\alpha_2 + b_4\beta_2 + ...b_{2n-1}\alpha_n + b_{2n}\beta_n$$

Now, we have to define the mapping $C : \Re^n \rightarrow \Re$ as the composition of FFT and the mapping B.

Let $f_1, f_2, ...f_n$ be the basis vectors in $\Re^n$.

We have

$$f_1 = (1, 0, 0, 0, ...0), f_2 = (1, 0, 0, 0, ...0), ..., f_n = (1, 0, 0, 0, ...0)$$

Any vector $x = (x_1, x_2, x_3....x_n)$, xi $\in \Re$ can be expressed as $x = \sum_{i=1}^{n} x_i f_i$

To define C it is sufficient to define numbers $C(f_i)$ for $i = 1, 2, ...n$. $C(x) = C(\sum_{i=1}^{n} x_i f_i) = \sum_{i=1}^{n} x_i C(f_i)$Say FFT$(f_i) = d^i = (d_1^i, d_2^i, ...d_{2n}^i)$Then $C(f_i) = (d_1^i b_1 + ...d_{2n}^i b_{2n}) \in \Re$

Denote $C(f_i)$ as C_i

Using these notations we obtain $C(x) = \sum x_i C_i \in \Re$

Thus the check for correctness in FFT computations is $C(x) = \sum x_i C_i \in \Re = B(y)$

Allowing for round-off errors this expression becomes:

$$B(y) - \sum x_i C_i \in \Re \leq \tau$$

where $B(y)$ is as derived before : $B(y) = b_1\alpha_1 + b2\beta_1 + b_3\alpha_2 + b_4\beta_2 + ...b_{2n-1}\alpha_n + b_{2n}\beta_n$ and τ is a variate that is determined by observing the round-off errors in our experiment runs.

5.1.1 Mathematical Proof

Expression $B(y) - \sum x_i C_i \in \Re = \tau$

can be expressed as

$$B(y) = Re(y.AL^*)$$

where $AL = (b_1 + \jmath b_2, b_3 + \jmath b_4, ...b_{2n-1} + \jmath b_{2n})$

FFT(x) = xD^T where D is the $n * n$ FFT transform matrix and $D = (d^1, d^2, ...d^n)^T$

Now applying the same transformation to FFT(x) we have:

$B(FFT(x)) = Re(x.D^T.AL^*)$

If x is real, this reduces to:

$B(FFT(x)) = xRe(D^T.AL^*)$

=

$$d^1.(b_1 + \jmath b_2, b_3 + \jmath b_4, ...b_1 + \jmath b_{2n-1}, b_1 + \jmath b_{2n})^*$$

$$d^2.(b_1 + \jmath b_2, b_3 + \jmath b_4, ...b_1 + \jmath b_{2n-1}, b_1 + \jmath b_{2n})^*$$

$$x.Re\{ \quad .$$

$$.$$

$$d^n.(b_1 + \jmath b_2, b_3 + \jmath b_4, ...b_1 + \jmath b_{2n-1}, b_1 + \jmath b_{2n})^*\}$$

We observe that $(d^i).(b_1 + \jmath b_2, b_3 + \jmath b_4, ...b_1 + \jmath b_{2n-1}, b_1 + \jmath b_{2n})^*$

is infact C^i and the above expression reduces to :

$$B(FFT(x)) = \sum x_i C_i \in \Re$$

Now taking a step back and assuming x is complex we have:

$$B(FFT(x)) = Re(x.D.AL^*) = Re(x).Re(D.AL^*) - Im(x).Im(D.AL^*)$$

Re$(D.AL^*)$ has already been represented as C_i. Im$(D.AL^*)$ can be represented as K_i:

$$K_i = (-1)d_1^i b_2 + d_2^i b_1 + (-1)d_3^i b_4 + d_4^i b_3.....(-1)d_{2n-1}^i b_{2n} + d_{2n}^i b_{2n-1}$$

Thus we have,

$$B(y) \approx \sum Re(x_i C_i) - \sum Im(x_i K_i)$$

We can see that this involves $4n^2 + 2n$ multiplies in all. Elements of the sets C_i and K_i can be computed offline. This reduces the number of multiplies to $4n$ "online" real multiplies. If x is real, then this reduces even further to $2n$ "online" multiplies.

5.2 EDM 2

Turmon et.al. [1] present an approach where the probe vector not only reduces the left-hand and right-hand sides to a scalar value, but also considerably reduces the number of computations involved. This becomes evident when we express Equation 1 as

$$w^T y - w^T W^T x = \tau$$

Evaluating wy^T involves n multiplies and evaluating $wW^T x^T$ involves $n^2 + n$ complex multiplies. Thus the total number of real multiplies involved are $16n^2 + 8n$ (assuming 1 complex multiply = 4 real multiplies). The component wW^T can be computed offline as it is independent of the input vector. This reduces the number of "online" multiplies to $8n$ real multiplies.

Turmon et.al go on to define a check for d :

$$|d|/(nlog_2 n||x||_2||w||_2) = \theta u \tag{2}$$

Where θ is an input-independent threshold

u is $2.2 * 10^{-16}$ and $||x||_2$ is the vector two-norm of x and $d = w^T y - wW^T x$ and $|d|$ represents the magnitude of the complex scalar d. We represent θu as δ.

The scaling by the matrix size results in an input independent threshold value. We shall refer to this algorithm as EDM 2. To determine τ, we run a series of experiments under fault-free conditions. This is discussed in the following section.

5.3 EDM 3

Another applicable error detection algorithm is based on Parseval's theorem which we shall refer to as EDM 3. EDM3 exploits the fact that the energy content of a waveform remains constant across a Fourier Transformation. In fact, this is true for all unitary transformations, i.e. where the transform matrix is a unitary matrix. ($A^{-1} = A^*$) as this property ensures energy conservation. Therefore, for Fourier Transform we have:

$$||y||_2 = \sqrt{n}||x||_2$$

We thus have a post-condition check

$$(||y||_2 - \sqrt{n}||x||_2)/||x||_2 \approx \tau$$

Evaluating this condition incolves $2n$ online multiplies.

CHAPTER VI

CHARACTERISTICS OF ERROR DETECTION MECHANISMS FOR FFT

In this chapter we define and describe the various characteristics that are important for an Error Detection Mechanism. We also compare these characteristics for the three EDMs.

6.1 Execution Time

Execution time is important for any error detection algorithm. The result checker should involve minimal numerical computations and must take as little time for execution as possible. Obviously the result checker must execute faster than the original FFT computing application.

Looking at the algorithms described above, it can be seen that EDM 3 is most numerically efficient. Next is EDM 1 as it involves $4n$ online real multiplies in the complex case and $2n$ real multiplies when the input is real. EDM 2 is most computationally intensive involving $8n$ online real multiplies.

Below is a plot of the execution time in our implementation for each of these algorithms with varying vector sizes. Assumption is that the transform matrices and the probe vectors can be computed a priori offline. Therefore their computation time is excluded from the results:

Thus we see that as the vector length increases, the difference between the execution

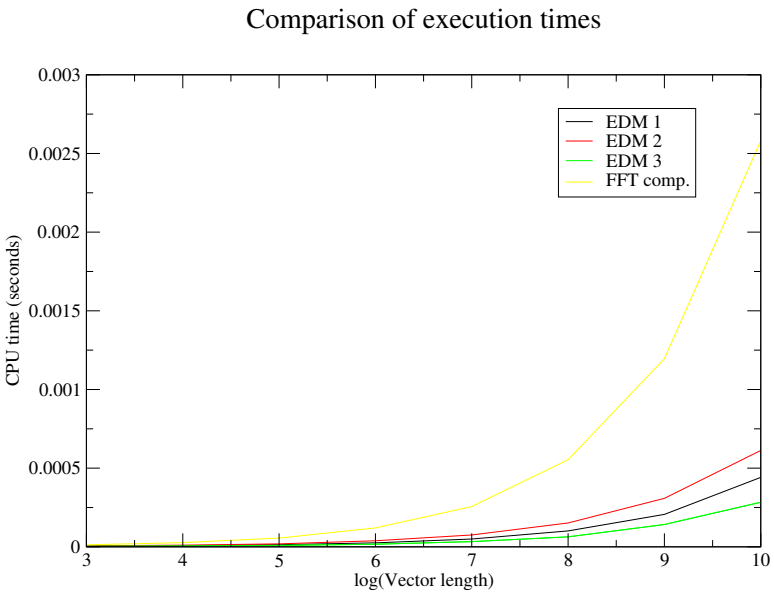

Figure 5: Execution times of the 3 EDMs for Complex input as vector length is varied

times of the three EDMs becomes more significant. We also observe that the execution times increase linearly with vector size(notice our x-axis is logarithmic).

In EDM 1, the number of online computations are reduced to half incase the input is real. Therefore it is interesting to compare the execution times in such a case. The plot below demonstrates this.

We observe that for real inputs, EDM 1 performs much better and has an execution time closer to EDM 3.

One is tempted to consider EDM 3 as an ideal choice. However, it is important to note that errors in sign bits are not detected by this algorithm. Errors in the sign bit mutate data in such a way that the magnitude of the data variable and thus the entropy of the signal remain unchanged. Therefore such errors in the input or output are not detected by this test. Errors in the sign bit are the most significant errors and cannot go undetected.

This leads to the other important characteristic of an error detection mechanism : its

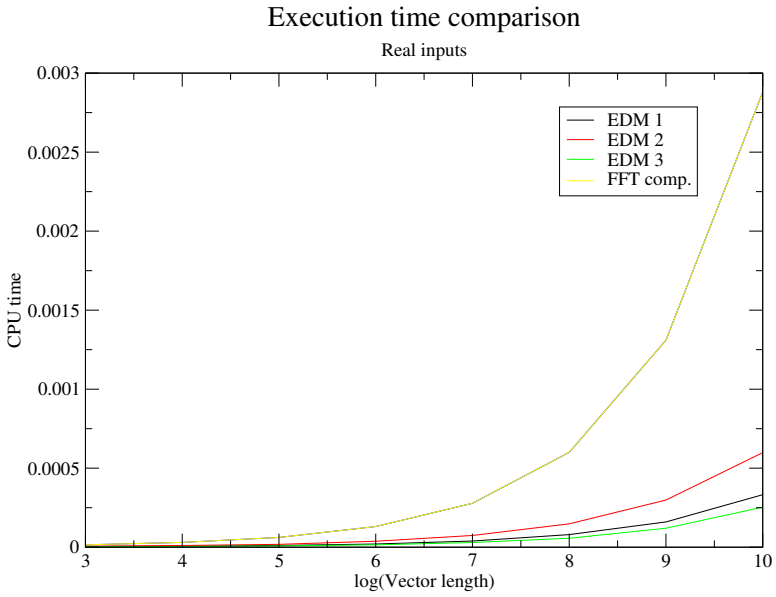

Figure 6: Execution times of the 3 EDMs for real input as vector length is varied

coverage. *Coverage* is defined as the probability of detecting a fault under a given fault probability distribution. Coverage can be estimated by the fraction of faults that are detected in a random sample of injected faults from the given distribution. In order for the error detection algorithm to detect maximum faults, it should have a high degree of sensitivity to faults. This implies that faults injected in the lower order bits must generate a high enough perturbation in τ so that they are detected.

In order to compare the error coverage of the three error detection mechanisms, we shall subject our test application to faults in each of the 64 bit representation at a random element of its data vector at various stages of the FFT computation and evaluate coverage by each mechanism.

6.2 Round off errors and Vector length

We study the variation of the difference in the post-condition check,τ that represents the round-off errors, for varying vector length for all the three error detection mechanisms.

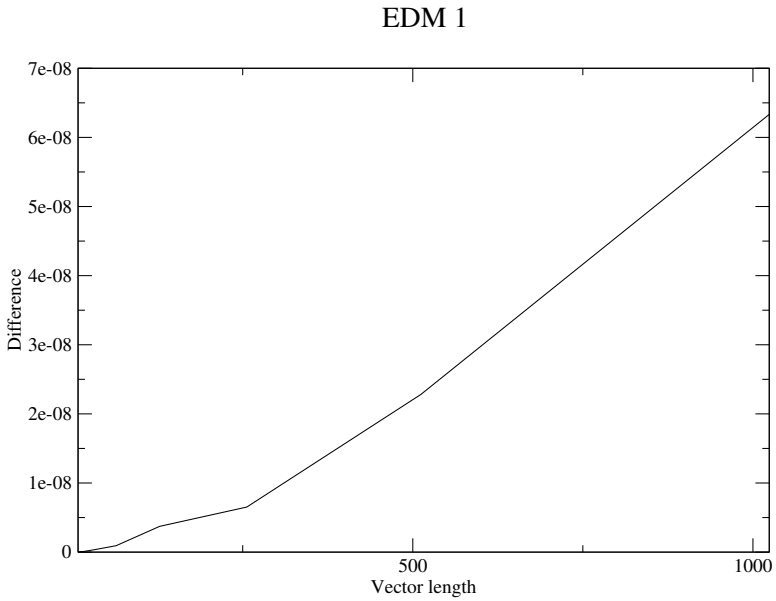

Figure 7: τ for varying vector length for EDM 1

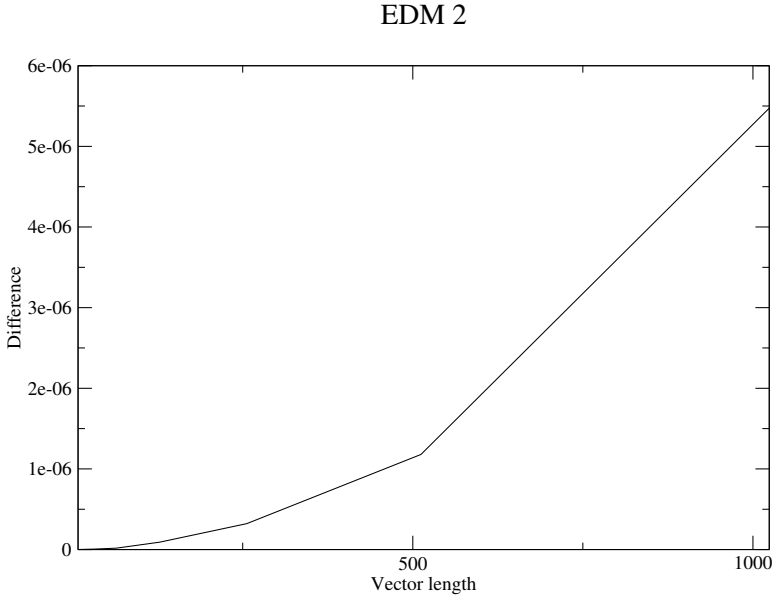

Figure 8: τ for varying vector length for EDM 2

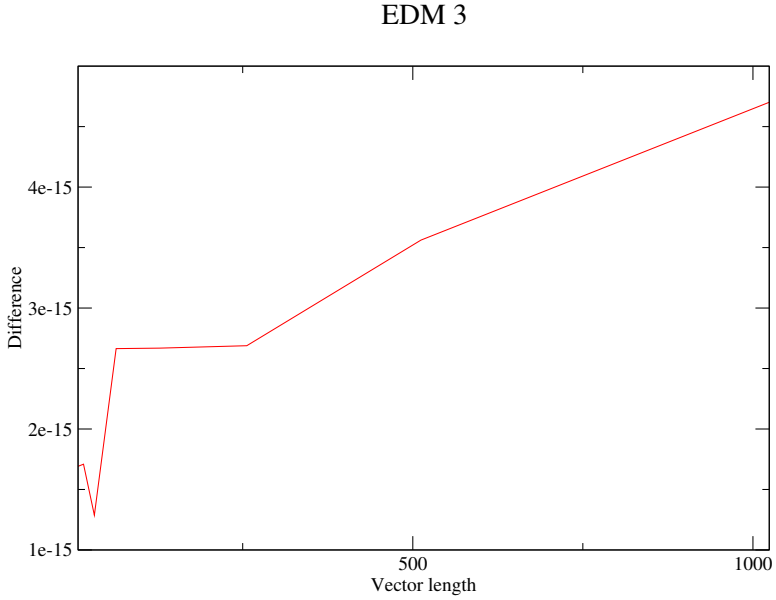

Figure 9: τ for varying vector length for EDM 3

We observe that τ for both EDM 1 and EDM 2 varies almost linearly with vector length while for EDM 3 this is a less strict relation.

We need to identify a suitable scaling for τ in order to make it independent of the input. We present analysis for EDM 1, although this can be used for the EDM 2 as well. We first divide τ by the length of the vector and observe its variation with input vector size.

From Fig. 6.6 There is a variation with input vector size. So we further divide this difference by $log_2 n$ where n is the vector size.

We observe that there is still a variation that is not constant over vector length(Fig. 6.7). We next divide this value with the norms of the input vector and the mapping vector, in order to make it independent of their magnitudes.

We see that this variation has a decreasing value with increasing vector size, which is to our advantage as it allows a tighter setting of thresholds(Fig.6.8) Finally, we observe the variation of the difference when the difference is scaled by the square root of the product

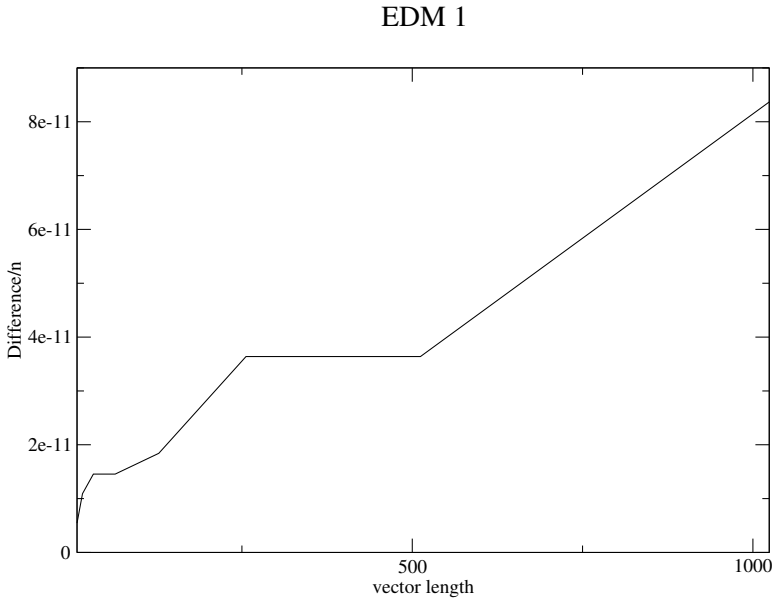

Figure 10: τ scaled by the vector length for varying vector length

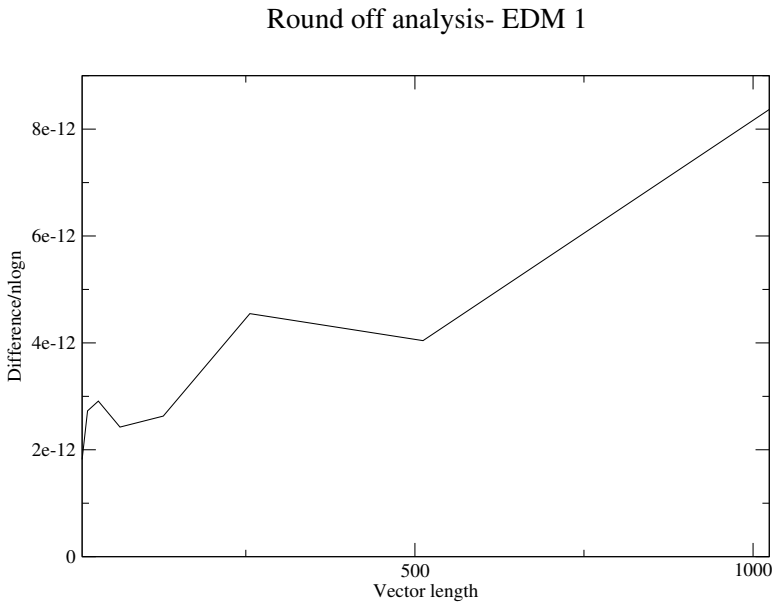

Figure 11: τ scaled by the product of vector length and its log to the base 2 for varying vector length

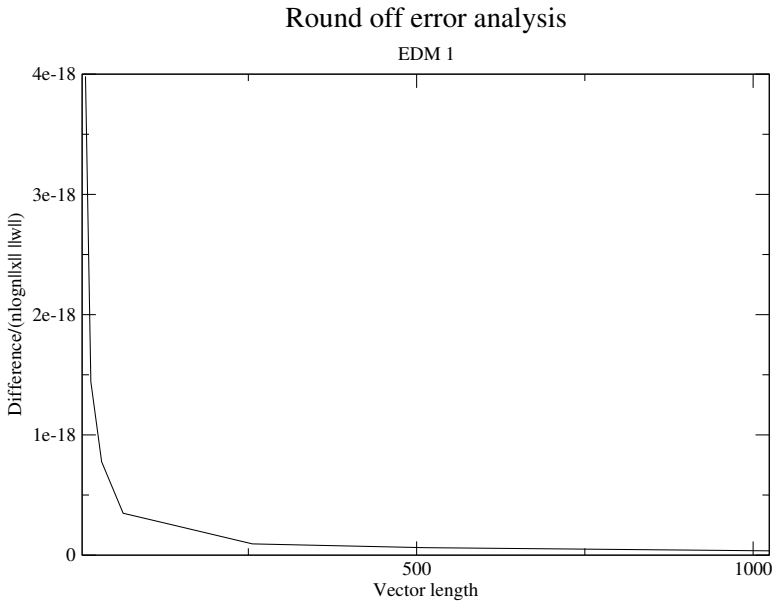

Figure 12: Magnitude of the difference in post-condition check scaled by $nlogn(n)||w||||x||$ for varying vector length

of the two norms.

From Fig.6.9 we observe that this is has a somewhat erratic behavior for vector sizes smaller than 256 although it becomes constant for larger vector sizes.

We observe that dividing the difference by the product of the square roots of the two norms in addition to $nlogn$ gives us the lowest possible thresholds.

Note that, this scaling is important only for setting thresholds so that we have thresholds independent of input vector magnitudes and length. Since we be divide both sides of our post-condition checks with the same scaling, we will not bias our round-off errors in anyway.

We thus adopt the product of vector length, norm of the input vector and norm of the mapping vector w as our scaling of choice. We shall denote this scaled difference in the post-condition check as *delta*.

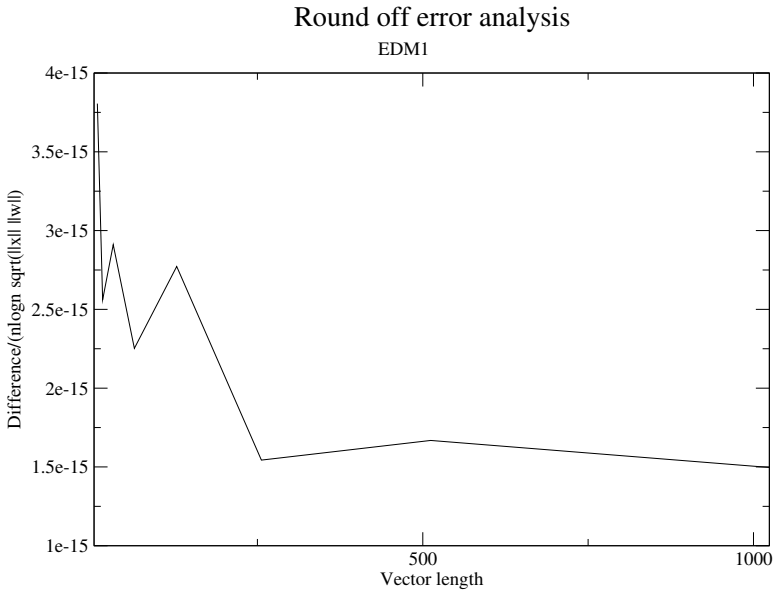

Figure 13: Magnitude of the difference in post-condition check scaled by $nlogn(n)\sqrt{||w||||x||}$ for varying vector length

6.3 Threshold values

An important step in implementing these error detection mechanisms is choosing a threshold value. Ideally, the threshold value must be independent of the vector length. In section 6.2 we determined a scaling for threshold values of EDM2 to make it independent of vector length. In this section we discuss the setting of threshold values and also study its variation with vector length for all EDMs.

Setting a threshold too low would mean a high false-alarm rate. A false alarm is when the error detection mechanism mistakes the round off error for a fault. The threshold must be chosen to be appropriately high to avoid false alarms at the same time low enough to detect as many faults as possible. It is required that the error detection mechanism has a high detection rate and low false alarm rate. In order for this to be true, the magnitude of δ should not vary heavily from one run to another. A small variation in δ across runs

enables us to set a low enough threshold. Lower thresholds would thus enable us to detect smaller variations in δ under faults and hence result in better coverage. In the following experiments we shall evaluate the variation of the threshold values in various runs. Setting of the threshold values is thus a compromise between high degree of error detection and low false alarm rate.

Finally it is required that the tolerances suggested by the error detection mechanisms be independent of the input vector length.

We have the following results for each error detection algorithm with varying vector length.

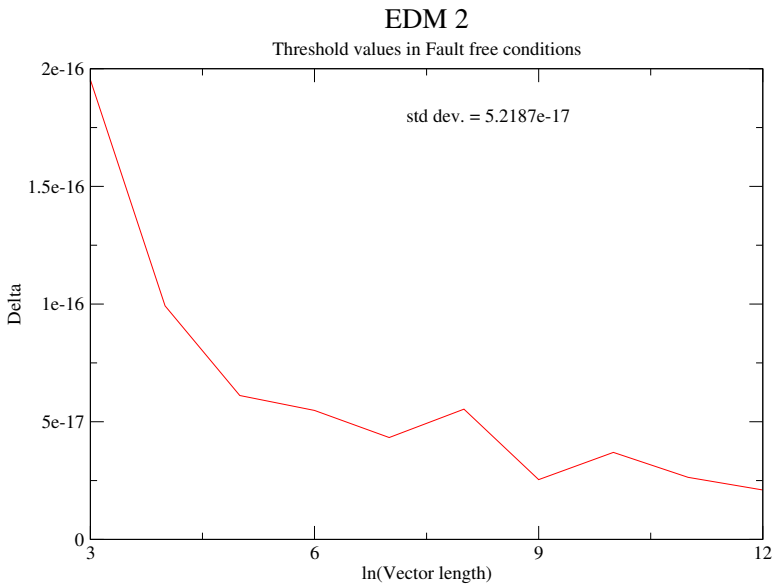

Figure 14: Threshold values under fault-free with varying vector length

We see that the thresholds are quite independent of vector length. Of the three, we see that EDM 1 has the minimum deviation across varying vector lengths. Next comes the EDM 2 followed by EDM 3. Moreover we see that as the vector length increases, the thresholds decrease for EDM 1 and EDM 2 where as for EDM 3 this increases.

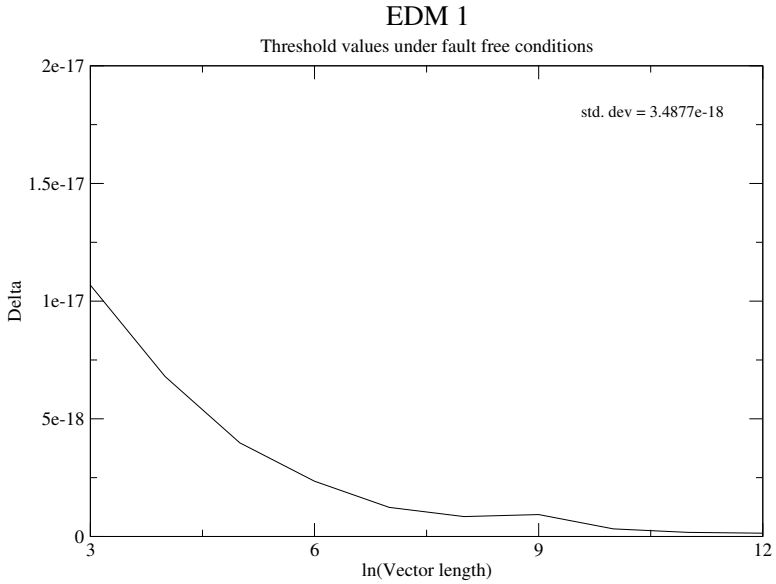

Figure 15: Threshold values under fault-free with varying vector length

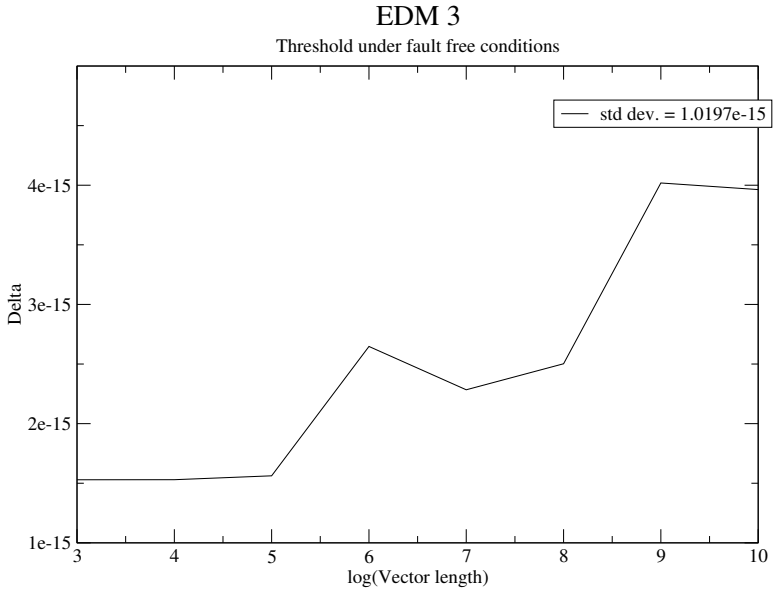

Figure 16: Threshold values under fault-free with varying vector length

A decreasing threshold value suggests better error detection if the threshold is re-adjusted for larger vector sizes. For EDM 3 the threshold value will have to be increased for larger vector sizes in order to avoid false alarms. This also means a decrease in error detection probability.

6.4 Relation between Threshold values and round-off errors

In this section, we evaluate the round-off errors in the FFT computation for our system. We then present the relation between this and the delta values we observed.

In order to compute the round-off errors inherent in our system that computes the FFT we proceed as follows:

Let x be the input vector and X be the FFT(x) computed by our system.

Then $x - IFFT(X)$ represents the total round-off error in the computation of FFT of x and its inverse FFT. (IFFT(X) represents the inverse FFT of X.) Assuming the worst-case that the round-off errors are cumulative, we approximate the round-off error in the computation of FFT as $(x - IFFT(X))/2$.

For vector lengths of 64, this round-off error was found to be of the order of 10^{-13}. The worst case round-off was found to be of the order of 10^{-12}.For our sets of input, the input has a range of 10^3 in magnitude and hence a norm of the same order. When the round-off error 10^{-12} is divided by the norm of the input and the norm of our w vector (fixed at the order of 10^2) we get a threshold of the order of 10^{-17}. This is in agreement with the worst-case threshold obtained for the EDMs.

For a higher dynamic data range of 10^7 in magnitude, the round-off error was found to be of the order of 10^{-10}. However, the norm of the input vector was also higher (of the

order of 10^7). resulting in a threshold value that is the same as before. Thus dividing the τ value by the norm of the input vector makes it independent of the magnitude of the input.

When the vector length was increased to 256, the round-off error was still at the order of 10^{-12}, showing that the round-off errors are not cumulative and tend to cancel out on an average case. Scaling τ by the vector length is thus a worst-case measure.

CHAPTER VII

EXPERIMENTAL RUNS

7.1 Experimental Setup

7.1.1 Setting of thresholds

In order to understand the behavior of the round-off errors, we first run a set of experiments under fault-free conditions and observer δ. We run each EDM under fault-free conditions for 1,000,000 runs for randomly generated vector inputs of length 64. The inputs are generated such that the real and imaginary parts have a gaussian distribution with unit variance. The input has a dynamic data range in terms of magnitude of 10^3. The data however has a variation up to the 15^{th} decimal place. Thus the data varies from 0 to 1000.999999999999999. Thus, looking at the just the numbers involved (discarding the decimal point) we have a variation in data from 0 to 10^{18}. In that sense the data has a variation of 10^{18} Table 7.1 below lists the standard deviation of the δ values under fault-free conditions for each of the EDMs.

We see that δ under $EDM1$ has the minimum variance across the 1,000,000 runs while $EDM3$ has the maximum variance. Table 7.2 below illustrates the maximum value of δ obtained for each of the EDMS in the 1,000,000 runs. This represents the worst-case

EDM 2	*EDM 3*	$EDM1^{*}$	$EDM1^{**}$	$EDM1^{***}$	$EDM1$
1.1143e-17	1.5752e-16	1.3995e-09	2.1867e-11	3.6445e-12	3.2264e-18

Table 1: Standard deviation of δ under fault free conditions : $EDM1^{*}$ represents τ for $EDM1$; $EDM1^{**}$ is τ/n; $EDM1^{***}$ is $\tau/logn$; $EDM1$ is $\tau/nlogn||x||||w||$

EDM 2	*EDM 3*	$EDM1^{*}$	$EDM1^{**}$	$EDM1^{***}$	$EDM1$
8.5805e-17	1.1405e-15	1.3504e-08	2.1100e-10	3.5167e-11	2.7883e-17

Table 2: Maximum value of δ under fault free conditions

EDM 2	*EDM 3*	$EDM1^{*}$	$EDM1^{**}$	$EDM1^{***}$	$EDM1$
2.1504e-17	1.8202e-16	1.8348e-09	2.8669e-11	4.7782e-12	4.2541e-18

Table 3: Mean value of δ under fault free conditions

round-off errors:

Table 7.3 illustrates the mean value of δ obtained for each of the EDMS in the 1,000,000 runs. This represents the average-case round-off errors:

We set our thresholds based on the maximum value of the round-off errors. In our experiments, the threshold has been set as 10% above the maximum value of δ from 1,000,000 different runs of randomly generated input vectors under fault free conditions.

7.1.2 Choice of w vector

EDM 1, involves the product $D.AL^{*}$ which is equivalent to computing the FFT of the w vector. Similarly EDM 2 involves computing the product wW^{T} which is the FFT of the w vector. Therefore it is important that the w vector not have an FFT that is sparse in order to ensure that none of the elements of the input or output vector get zeroed out. This automatically eliminates the simplest mapping: a vector of all 1s or its multiples. We therefore construct a w vector whose real and imaginary elements have a normal distribution with unit variance. This ensures that the elements of w and its Fourier transform do not vary significantly in magnitude and therefore all elements of the vectors under scrutiny get weighed somewhat equivalently.

0	*1-11*	*12-63*
Sign	Exponent	Significand

Table 4: IEEE representation of Double precision numbers

7.1.3 Input Vectors and Error Analysis

The input vectors are randomly generated based on a normal distribution with unit variance. The data is generated such that it has a dynamic deviation of 10^3 in magnitude of its whole number portion and a dynamic range of 10^{18} in all.

We run our experiments for a million runs each. For a sufficiently large sample size n, the measured probability of detection p has approximately a normal distribution[29]. The standard error of the probability estimate p is given by $\sqrt{p(1-p)/n}$ where $np \geq 10$ and $n(1-p) \geq 10$ [30]. Thus for all probabilities greater than 0.00001 or less than 0.99999 we have a standard error that is less than 0.0005.

7.1.4 Fault Injection Experiments

The IEEE 64-bit representation of a Double-precision number is illustrated in Table 7.4.

The first bit is the sign bit followed by 11 bits of the exponent and finally 52 bits of the significand. The double precision number is represented by: $(-1)^s(1 + .significand_{two}) * 2^{exponent-1023}$ The sign bit and the exponent bits are therefore the most prominent bits. A bit flip introduced in these bits leads to maximum mutation of data. Please see appendix A for an illustration of the effects of bit flips in the magnitude of a data element.

The first set of our experiments are designed so as to evaluate the error coverage of all the three error detection mechanisms. Faults are injected at each bit of the 64-bit representation of double precision vectors at randomly chosen vector elements. The fault injection

is timed to occur at the beginning of the $logn$ stages of the FFT computation.

Since we are considering vectors of length 64, our coverage experiments constitute injecting single bit flips at each of the 6 stages in the FFT computation.

7.2 Experiment Runs

The first set of experiment runs are designed to evaluate the coverage of the EDMs. For this we measure the number of detected faults in a million runs, with each of the 64 bits being corrupted one by one. Bit flips are introduced at all the 6 stages of the 64-length FFT computation into the data array. Faults at the input stage would fan out to all elements of the FFT while single bit flips at the final stage might effect only two elements of the final output. Thus we determine coverage for all stages of the FFT to study such cumulative effects of faults.

7.2.1 Faults at input

We have the following observations of error detection for faults at the input stage. The x-axis represents the number of the bit being flipped and the ordinate represents the probability of error detection. The probability of detection of an EDM is computed by measuring the number of errors detected by the EDM in a million faulty runs divided by the number of runs.

We see that errors in the Sign bit are not detected by EDM 3. Also, as bit flips start getting less significant, the error detection capability of EDM 3 starts reducing first. EDM 2 and EDM 1 algorithms are very close to each other initially, but for bit flips at bit numbers in the 50's, EDM 1 is better. While EDM 3 does not detect all errors starting from bit 30, its error detection capability reduces a lot slower than that of EDM 2, as the significance of the injected faults reduces. We observe that EDM 1 has an overall error detection capability that is better than the other two EDMs.

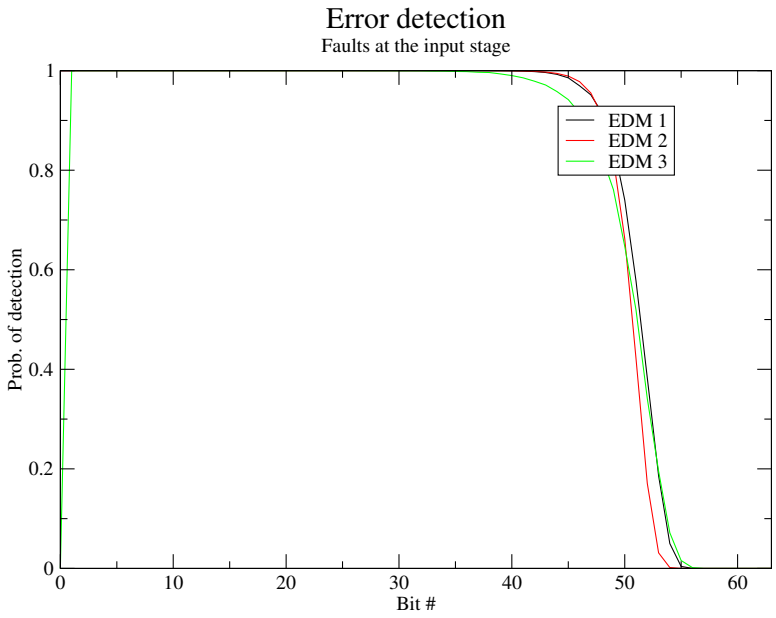

Figure 17: Prob. of detection for faults at the input stage of FFT

7.2.2 Faults at various stages of FFT

Faults are injected at the intermediate stage of the FFT computation. Since ours is a 64 length vector, we have 6 stages. We corrupted data in the input stage, in the previous section. We now inject faults at the inputs of stage 2 to stage 6 and finally the output of stage 6, which is also the output of the FFT.

We observe that the error detection probability does not vary considerably for faults injected across the various stages of the FFT.A further discussion to explain this is carried out in Section 7.2.8.

7.2.3 Faults at random bits

This sets of experiments is geared towards evaluating the error detection capability of the EDMs under real-life conditions where faults can occur randomly at any bit. Here we inject faults at random bits of the data array at each of the 6 stages of the FFT.

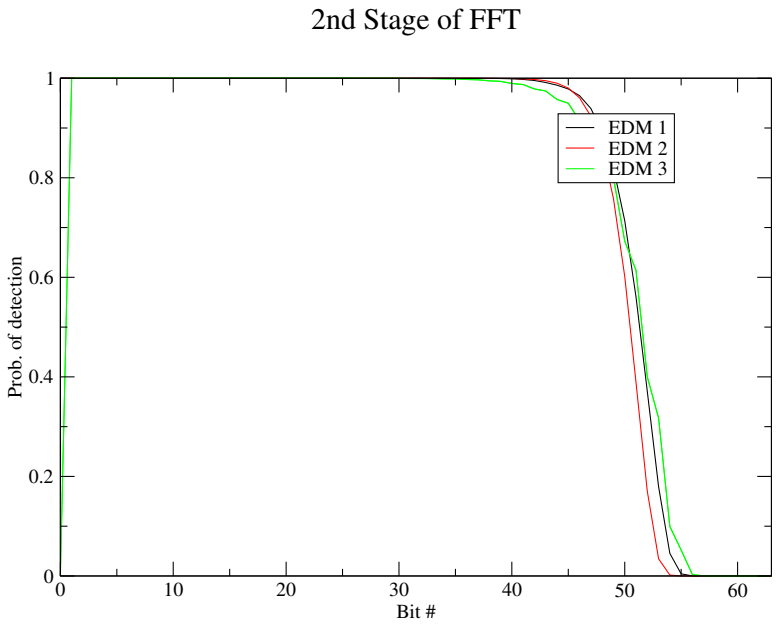

Figure 18: Prob. of detection for faults at the 2^{nd} stage of FFT

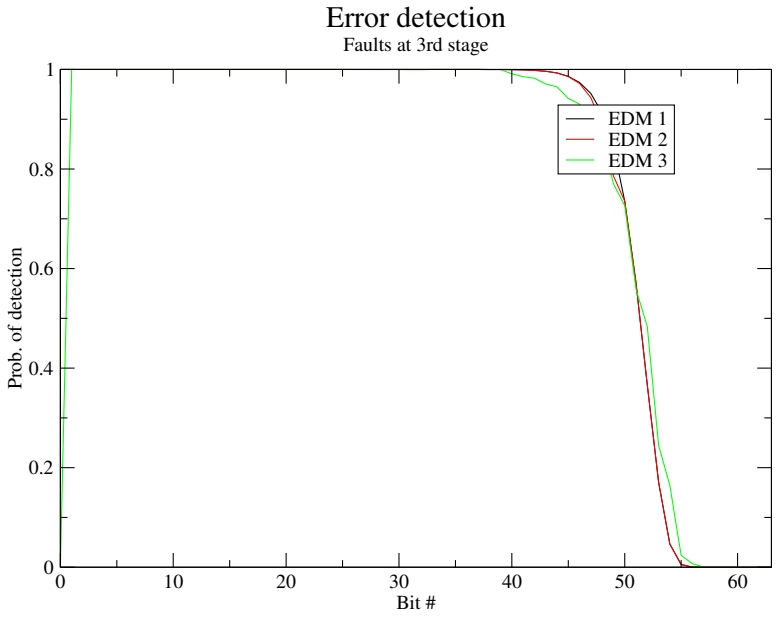

Figure 19: Prob. of detection for faults at the 3^{rd} stage of FFT

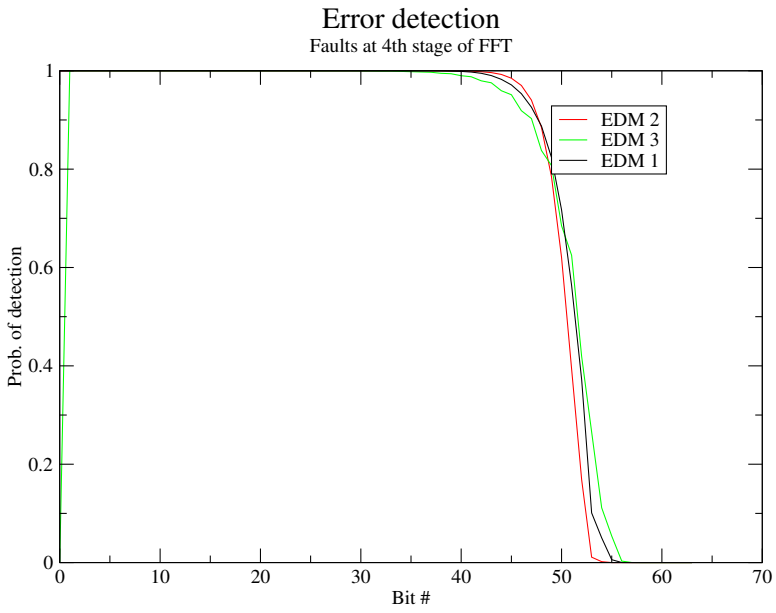

Figure 20: Prob. of detection for faults at the 4^{th} stage of FFT

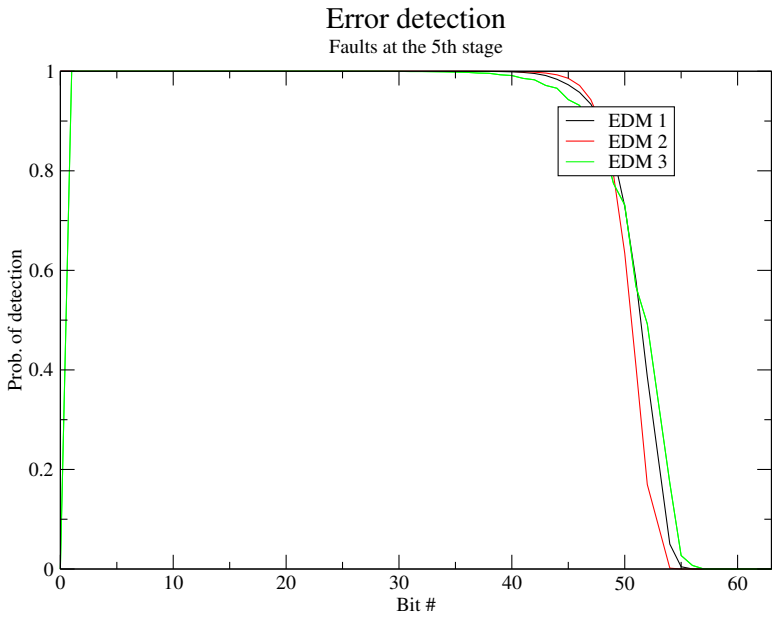

Figure 21: Prob. of detection for faults at the 5^{th} stage of FFT

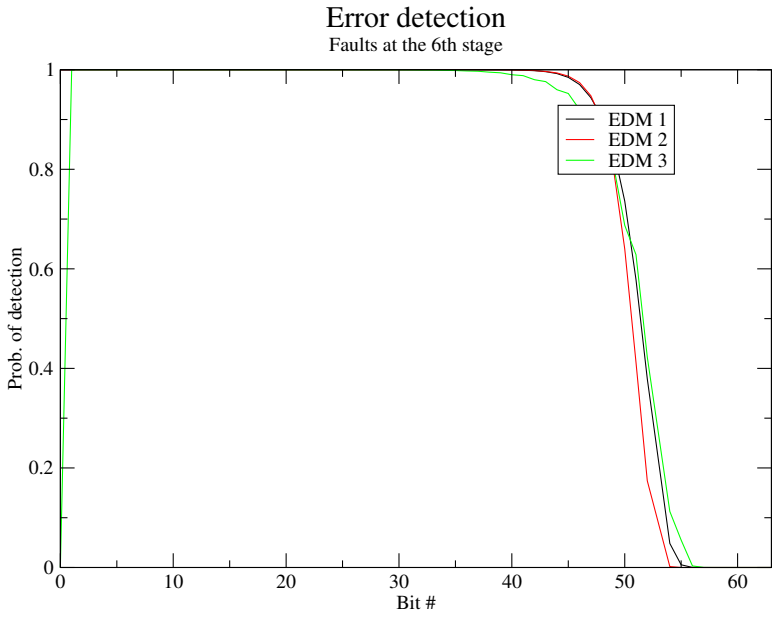

Figure 22: Prob. of detection for faults at the 6^{th} stage of FFT

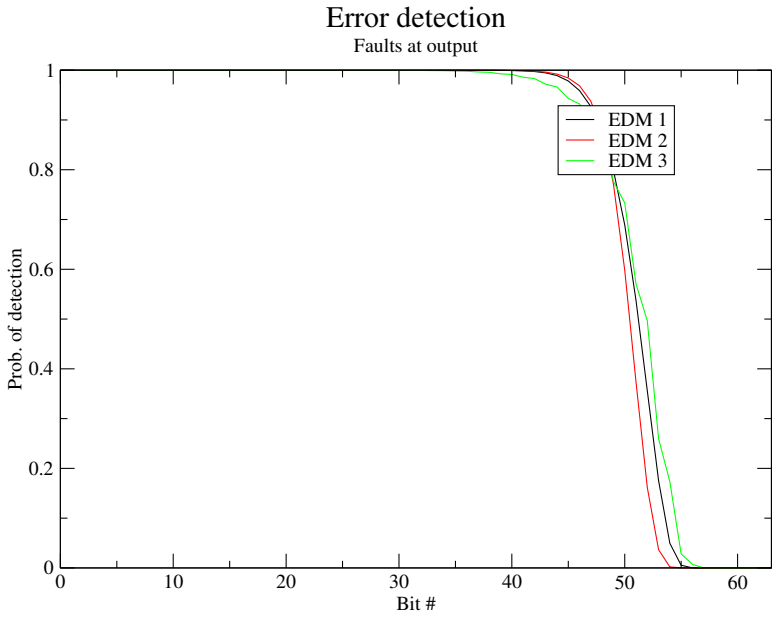

Figure 23: Prob. of detection for faults at the output stage of FFT

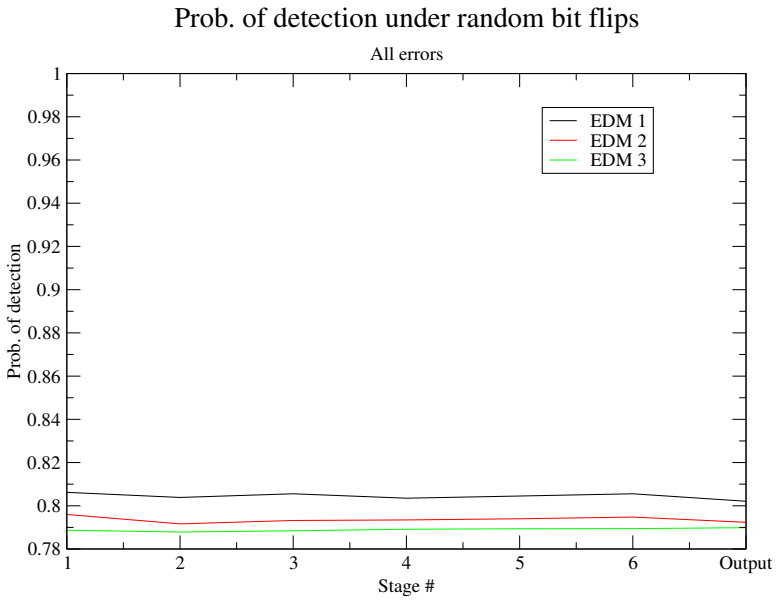

Figure 24: Prob. of detection for random bit flips, all faults included

We observe that EDM1 has a better error detection capability, followed by EDM 2(Fig. 7.8). Moreover, the error detection capability for random errors remains the same across faults at various stages.

However under random bit flips there exist faults that are more severe than others and error coverage is not a true measure of the error detection capability for such faults. For example, faults in the Sign bit go undetected by EDM 3. When we measure the coverage, a fault that goes undetected in the sign bit is treated equivalent to say a fault that goes undetected in the 62^{nd} bit. Therefore, in order to evaluate the true error detection capability we exclude such small errors from our measure.

We run a set of million runs. For each of these runs, we introduce a random bit flip in the data array. We then, determine if this error is a significant one. Since we are interested in the effect of the bit flip rather then the bit-flip itself, we consider the significance of the error introduced in the output. This is done by computing the FFT of the same input vector under

fault-free conditions and determining if the difference between the two computed outputs is significant. The set of experiments below discount all relative errors in the output that are less than 10^{-10} from the coverage i.e. if y is the output element under fault-free conditions and *Y* is the corresponding output element computed under faults, we consider only errors that result in (y - *Y*)/y $> 10^{-10}$. The reason for choosing 10^{-10} is because this is reflective of the round-off errors in the system.

It can be argued that this significant value of error can be chosen to be much greater, and closer to the quantization noise in the system. This is because errors errors smaller than the quantization noise are indistinguishable for the analog to digital signal converter. In that case, assuming the level of quantization for our data range is of the order of 10^4 or larger,

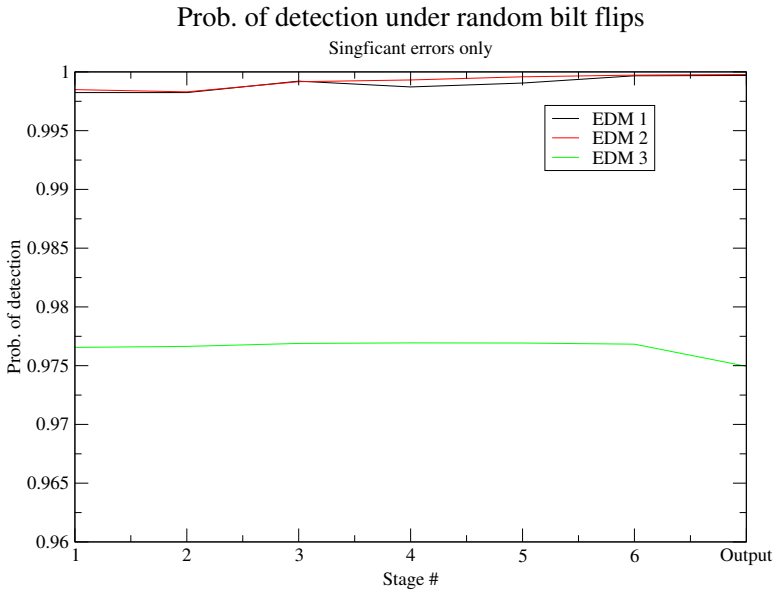

Figure 25: Prob. of detection for random bit flips,for faults that lead to errors in computed output $> 10^{-10}$

We see that for significant faults, both EDM 2 and EDM 1 have a prob. of detection very close to 1. However EDM 3 performs poorly in comparison. The poor performance

is mostly due to its inability to detect Sign bit faults. We observe that EDM 3 has a probability of detection less than the other two EDMs by approximately 0.02. This value can be justified by arguing that faults greater than 10^{-10} occur between bit 0 and around bit 44. This means 1 in every 44 faults is not detected by EDM 3. This results in a probability of not being detected of 0.02.

7.2.4 Special Cases

We now evaluate the performance of the EDMs under a set of worst-case inputs. Worst case inputs from an error-detection perspective include input signals with all energy concentrated in one frequency bucket, or signals whose magnitude has a high peak at a time instant and very low otherwise. For such signals, there is a high potential for catastrophic cancellations. For instance when a very high value is added to an extremely small value, the small value may be lost as a round-off error. Now when an equally high value is subtracted again from this, this might cancel out the first high magnitude and the small value is entirely lost. Thus the order of such computations contributes significantly to the round-off errors.

Our experiments constitute generating such waveforms and evaluating the error detection capability of the three EDMs under these conditions.

7.2.5 Faults at input

We generate a set of inputs such that the vector consists of one or two values with magnitude greater than 10^9 and the rest of the values around 10^{-3}. Faults are injected at random locations in the data array at the input stage. Faults here constitute random bit flips. The error detection capabilities of the three EDMs is evaluated under a million runs. The graph

below plots these values. An important consideration here is the understanding what magnitude errors are significant. Since our data has a variation from 10^{-3} to 10^{9}, we have three sets of probabilities with errors smaller than 10^{-5}, 10^{-4}, 10^{-3} discounted respectively.

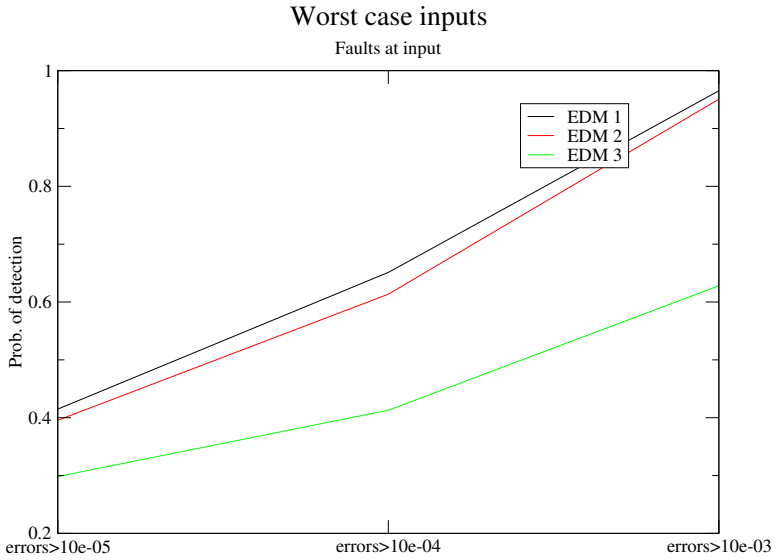

Figure 26: Prob. of detection under worst case inputs, faults at input stage

7.2.6 Faults at output

Here we are interested in a set of inputs where the energy content is extremely high in one frequency bucket. For this we generate inputs whose Fourier Transform has a singly extremely high magnitude with the rest of the magnitudes being very low. Experiments constitute of generating such vectors and taking their inverse Fourier Transform. Faults are then injected at the output stage of the FFT.

The graphs below demonstrate the error detection capabilities under such conditions. We observe that the error detection capability is reduced considerably under these set of inputs. EDM 1 performs better among the three where as EDM 3 performs the worst in both cases.

Figure 27: Prob. of detection under worst case inputs, faults at output stage

errors	*EDM1*	*EDM2*	*EDM3*
All errors	0.7815640	0.7717160	0.7399770
Significant errors	0.9985968	0.9988590	0.9530402

Table 5: Prob. of detection for a dynamic data range of 10^{22}

7.2.7 Larger dynamic variation of data

So far we have been considering inputs with a dynamic data variation of 10^{18}. We now consider a set of experiments where the input has a larger dynamic data variation of 10^{22} in terms of the whole number portion and 10^{22} in all. We do not expect the δ values under fault free conditions to vary considerably, as we divide the round-off errors by the norm of the vector to make it input independent.

Table 7.5 illustrates the probability of detection for this data under random bit flips at input. The probability of detection under similar faulty conditions for smaller dynamic data variation of 10^{18} is also shown for comparison in Table 7.6.

We see that EDM 3 performs much poorer than the other two algorithms. For the other two EDMs the detection capability does not vary too much with the change data dynamic

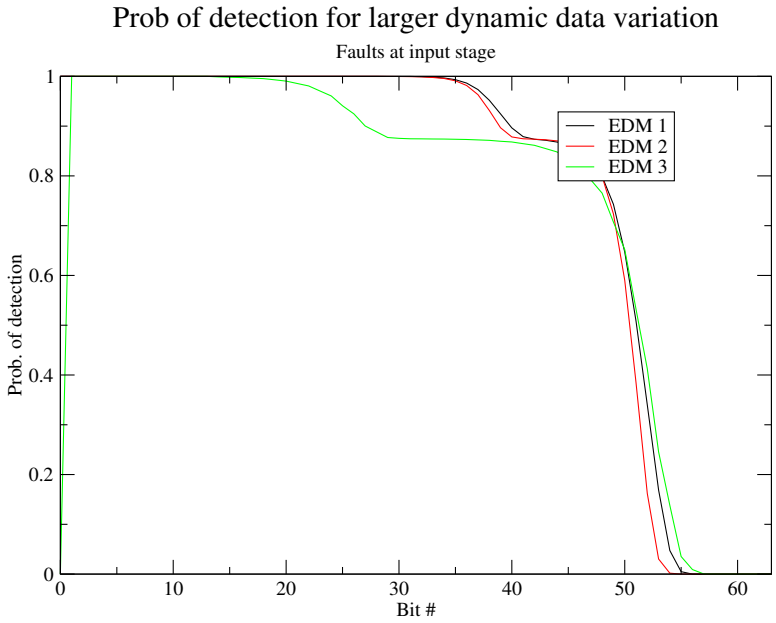

Figure 28: Prob. of detection for a dynamic data variation of 10^{22}, faults at input stage

errors	*EDM1*	*EDM2*	*EDM3*
All errors	0.8061980	0.7959760	0.7887210
Significant errors	0.9982422	0.9984893	0.9765542

Table 6: Prob. of detection for a dynamic data range of 10^{18}

range, however the detection probability of EDM 3 is considerably reduced. Looking at the coverage characteristics in Fig. 7.12 we observe that EDM 3 has a distinctly poorer detection probability for higher dynamic data ranges. We see that although the bit-wise error detection for a larger dynamic data range is poorer, for random bit flips and significant faults the performance of EDs 1 and 2 are not very different for the two sets of data ranges. This can be explained as follows. In case of large data range, for data elements with very large magnitudes a bit flip say in bit number 20 has a much higher impact on its

magnitude than the same bit flip in a data element of smaller magnitude. But a small change in magnitude in this small data element has a much lower impact on the computed output (since this small magnitude element is added or subtracted to a much larger magnitude element and the change in its magnitude due to the error becomes a lot less significant) as compared to the smaller data range case. Since we are interested in errors in the output to determine if the injected faults are significant, the case described above is not considered. And when the same bit flip is introduced in a larger data element, the effect is a large change in- input and hence the output and this is easily detected by the EDMs. Thus the probability of detection of significant errors remains around the same for the two EDMs. For EDM 3, from Fig. 7.12, we see that from around bit number 20, it has a prob. of detection less than 1. Assuming, significant errors occur from bit number 0 to 23(average found from a million runs), we realize that 1(Sign bit) in every 23 bit flips is not detected.

7.2.8 Error Propagation across FFT stages

We observed above that the error detection does not vary too much across the various stages of the FFT. In order to better understand this, we consider the 8x8 FFT. It can be recounted

that the first element of the FFT of a vector is equal to the sum of the magnitudes of the input vector elements. This element, which is also called the D.C component of the FFT, has the highest magnitude as compared to the other elements of the FFT vector. Errors at different elements have different impact on the computed output. It can be easily proved that a constant error in the input vector will produce a constant error in the output vector. ($FFT(X + e) = FFT(X) + FFT(e)$, where e is the error in the input and X is the input vector). Thus the probability of error detection for faults injected at the input stage depends on the magnitude of error in the input. However for faults injected at the intermediate stages of FFT, depending on the element number and whether it is real or imaginary, it is scaled by a specific twiddle factor. Thus it is not necessary that a fault injected at an intermediate stage will have a proportionate effect on the output.

We study the behavior of errors under such conditions. For each stage of the FFT, we corrupt all elements of the 8 element vector. We plot the δ values computed by the three EDMs. We also determine the magnitude of error introduced by computing the difference between all elements of the computed output and all corresponding elements of the output under fault-free conditions. The norm of this difference vector represents the magnitude of the error. A fixed bit location of 39 was chosen for these experiments because it has prob. of detection less than one but large enough (based on our earlier experiments).

We see that faults at the input are always significant. Faults in the first element are also always significant for all stages. The maximum error magnitude does not vary considerably across the various stages. Thus although errors in the input stage are higher than the other stages, errors in the intermediate stages are not significantly different from each other. Errors at the output stage also have more constant magnitudes (or common magnitudes).

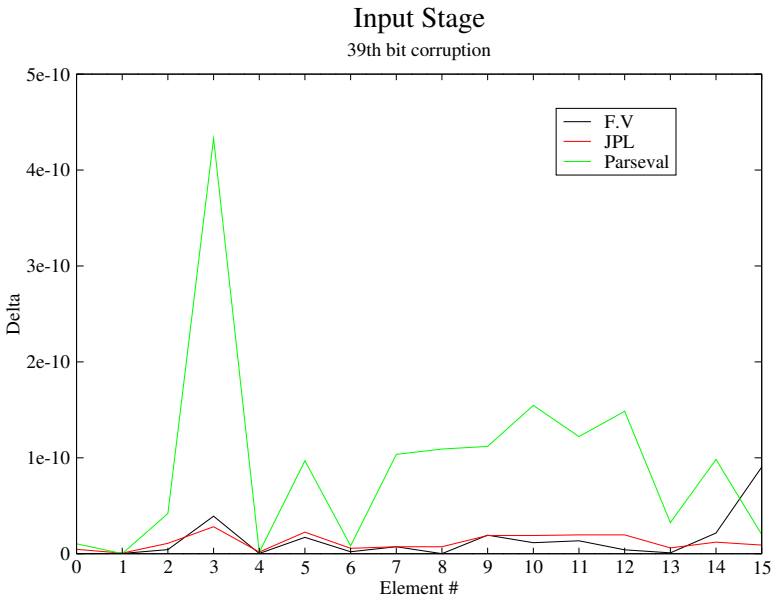

Figure 29: Variation of delta for 1x8 FFT; injection in each element at stage 1

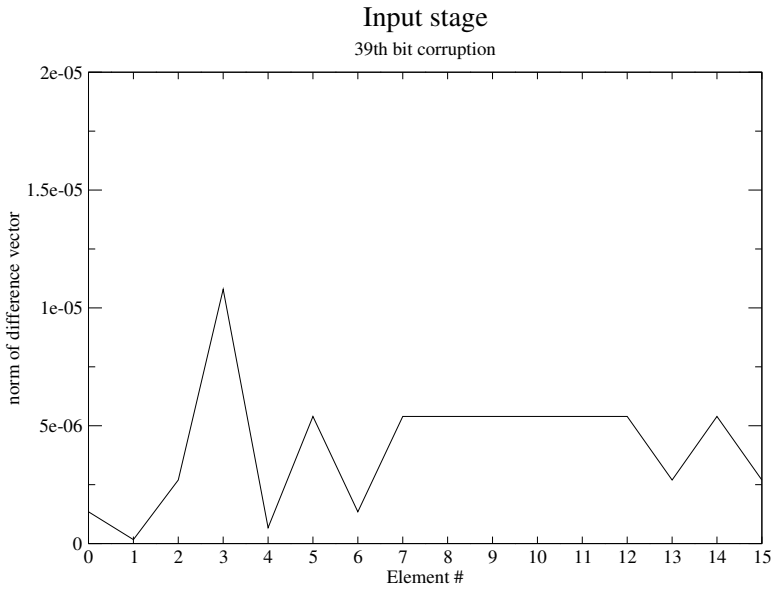

Figure 30: Variation in the difference between computed output in fault-free and faulty conditions of for 1x8 FFT for faults in each element at Stage 1

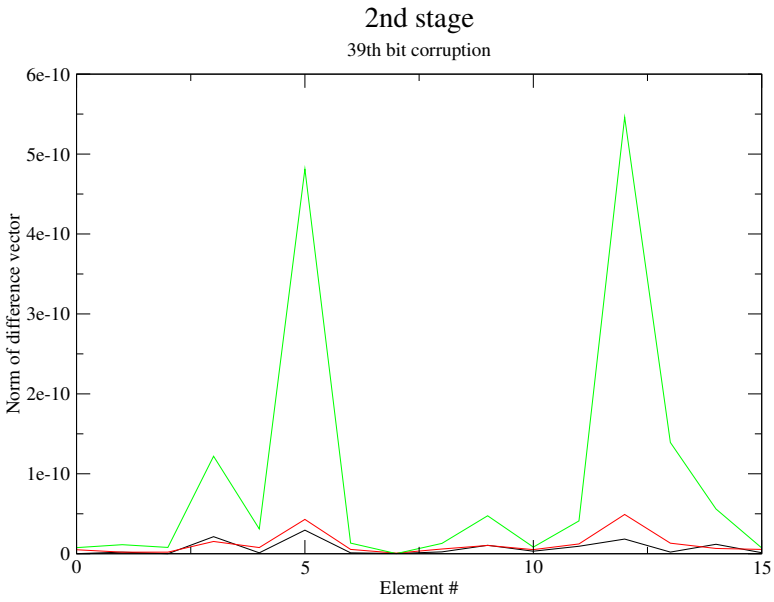

Figure 31: Variation of delta for 1x8 FFT; injection in each element at stage 2

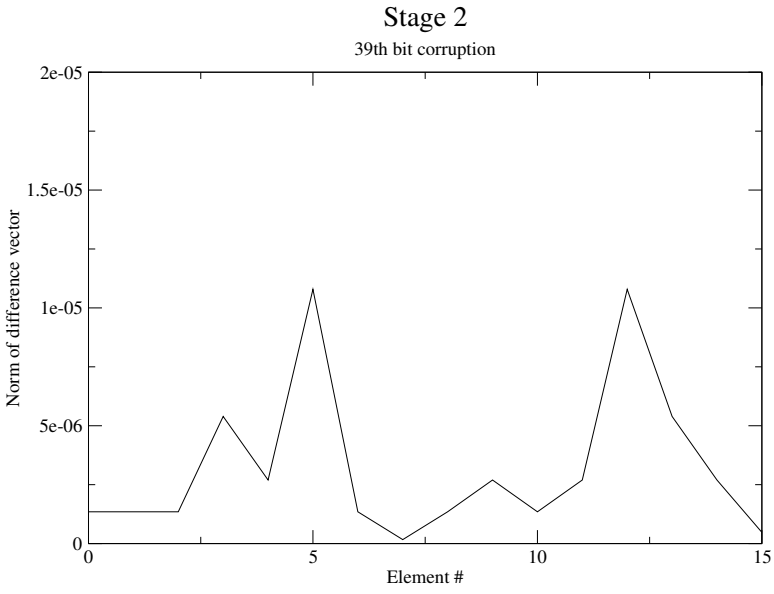

Figure 32: Variation in the difference between computed output in fault-free and faulty conditions of for 1x8 FFT for faults in each element at Stage 2

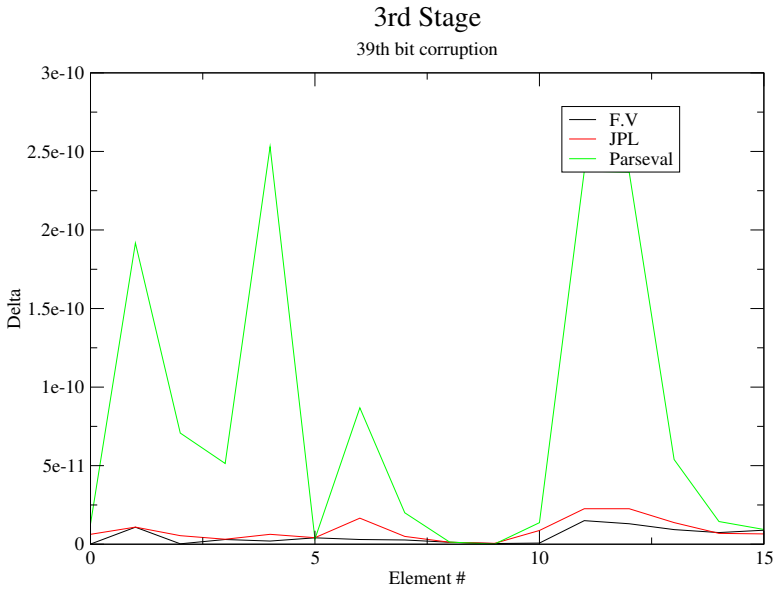

Figure 33: Variation of delta for 1x8 FFT; injection in each element at stage 3

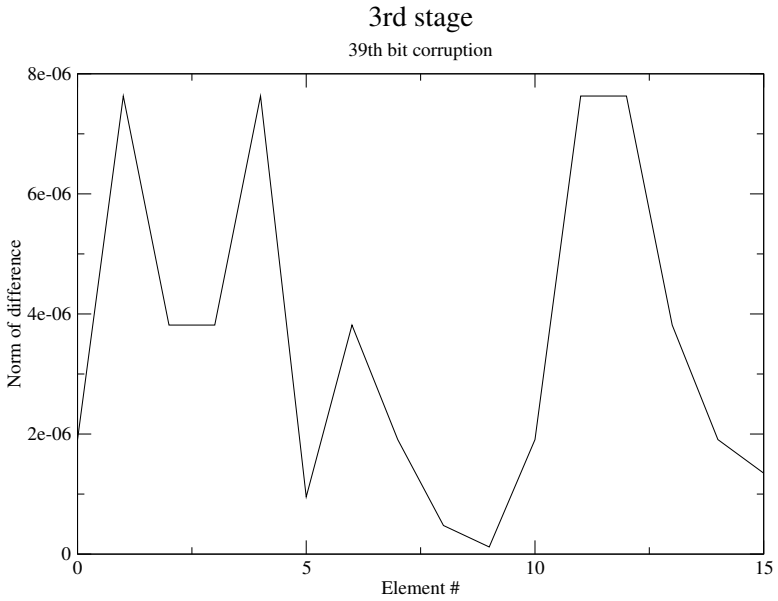

Figure 34: Variation in the difference between computed output in fault-free and faulty conditions of for 1x8 FFT for faults in each element at Stage 3

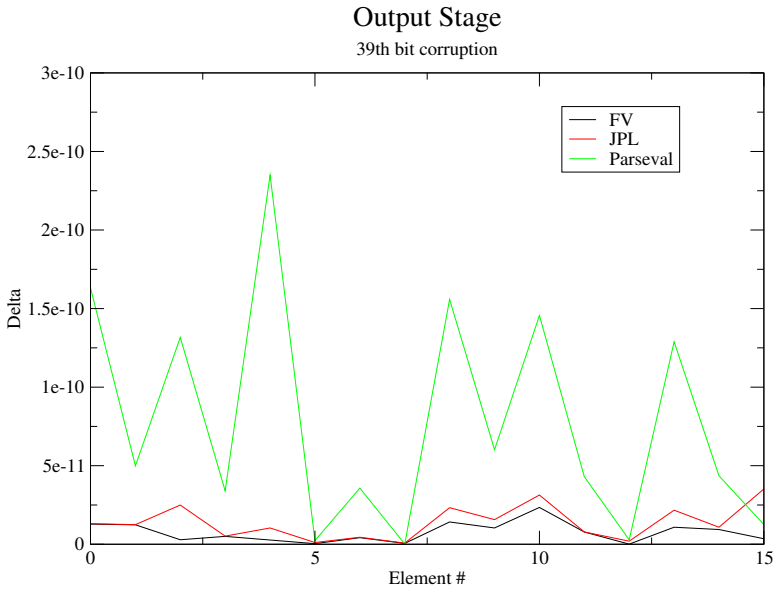

Figure 35: Variation of delta for 1x8 FFT; injection in each element at Output stage

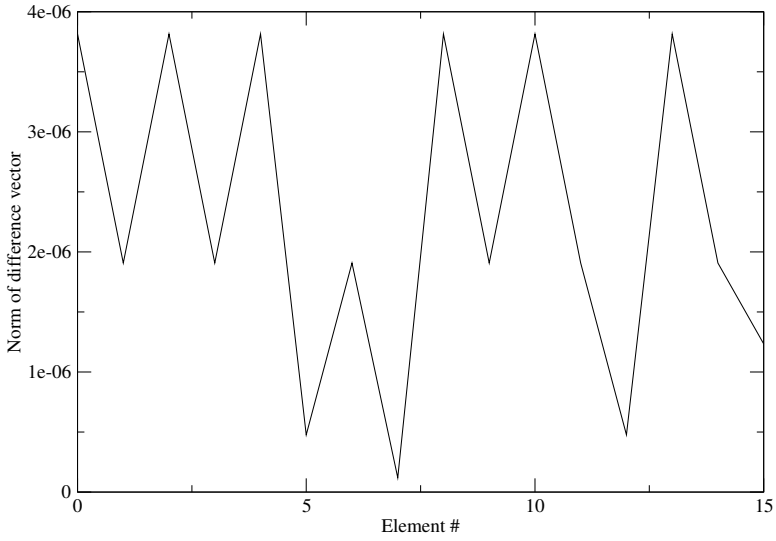

Figure 36: Variation in the difference between computed output in fault-free and faulty conditions of for 1x8 FFT for faults in each element at Output Stage

This is because there is no scaling by twiddle factors involved at the output. Obviously a fault in the output stage effects only the element that is being injected and there is no further impact on the other stages or other elements of the output vector. Since we are introducing a fixed magnitude error in all these EDMs, the error magnitudes are equal in many cases and hence we see a number of elements in the difference vector that are equal to each other (since the difference vector represents the magnitude of error introduced).

CHAPTER VIII

CONCLUSION AND FUTURE WORK

This thesis involves studying and comparing the existing approaches for Numerical result checking under memory faults and implementing a fault injector. The fault injector was used to simulate faulty conditions for the FFT applications under study.

We observe that EDM 1 has an error coverage that is equal to EDM 2 while taking half the time to execute. Although,for complex inputs, it takes a higher time to execute than EDM3 it has an error coverage that is highly superior to EDM 3. Thus EDM 1 distinctly emerges as the ideal choice of an Error Detection Mechanism, based on our experiments.

While the fault injection is a tool capable of injecting different kinds of faults for our study we restricted it to injecting faults into the heap of the application. This is because our FFT application was storing data in dynamically allocated memory space. By choosing a specific fault location in the data array (via API calls), we ensured that we have no latent errors, i.e all injected faults do manifest into errors.

8.1 Relation between the three EDMs

The relation between the 3 EDMs is as follows. EDM2 is obtained from multiplying the post-condition check with a probe-vector, also called the mapping w. EDM 1 can be interpreted as the real-part of this post-condition check difference. EDM 1 introduces the mapping in such a way that both the real and imaginary parts of the input and output are

considered and we obtain a scalar that reflects the real-part of the computation. EDM 2 computes the real as well as imaginary part of the post-condition check. This can be better explained with an example. Say we are interested in the accuracy of the computation $(a + jb)(c + jd)$. $(a + jb)$ is analogous to the input vector and $(c + jd)$ is analogous to the mapping or the probe-vector w. Now any error in the input $(a + jb)$ is reflected in both the real part of the output which is $a.c - b.d$ as well as the imaginary part of the output which is $a.d + b.c$. Thus by being interested in only the real part of the computation, EDM 1 has cleverly reduced the number of computations by half while retaining the detection capability. This can be seen in the detection probability of the two EDMs which are very close to each other for all our experiments.

EDM 3 can be considered as equivalent to EDM 1 with a mapping vector equal to the input vector itself. Thus the equation $B(y) = Re(y.AL^*)$ reduces to $B(y) = Re(y.y^*)$. The RHS equation changes from $Re(x.(FFT(w))^*)$ to $Re(x.(FFT(y))^*)$ or $Re(x.x^*)$ since $FFT(x^*) = (FFT(x))^*$

8.2 Summary of Results

We set the thresholds for each EDM based on a million runs under fault-free conditions. The thresholds were scaled to make them independent of input vector magnitudes and vector length. It was found that EDM 1 and EDM 2 had lower thresholds than EDM 3. Also EDM 1 and EDM 2 had smaller variation of the threshold values with varying input vector length. It was shown that all the three EDMs execute faster than the FFT computation algorithm. This was accomplished by pre-computing some input independent data offline and thus reducing the number of online multiplies. We showed that EDM 1 can

be reduced to $4n$ online multiplies for complex input and $2n$ multiplies for real input. EDM 2 can be reduced to $8n$ online multiplies and EDM 3 involves $2n$ online multiplies.EDM3 was the fastest algorithm followed by EDM 1 and finally by EDM 2. The execution times reported in this report also included the execution time for scaling τ to obtain δ.

We then evaluated the coverage of each of these EDMs and found that EDM 3 had the worst performance owing to the fact that is incapable of detecting sign bit errors. For a bit-wise corruption of data, it was found that the probability of detection starts falling below from approximately bit number 40 for EDM 1 and EDM 2 while for EDM 3 it starts falling from around bit number 30. The poorer performance of EDM 3 is more obvious in the case where we considered only significant errors for random bitflips in the input. We considered the significance of a fault in terms of the output computed rather than the input because we are interested in the impact of a fault rather than the fault itself. Round-off noise was estimated by evaluating the difference between the input vector and the inverse FFT of its FFT. It was found that for errors of the order of $10^{-}10$ and smaller, this difference was equal to the round-off noise in the fault-free case.

For significant faults, EDM 1 and EDM 2 had a very good probability of detection of around 0.998 while EDM 3 had a detection probability of 0.976 for faults in the input stage.

Based on our performance results, we see that while EDM 1 and EDM 2 detect faults upto the 40^{th} bit, EDM 3 starts having a prob. of detection less than 1 from around bit number 30. Thus it can be inferred that not only must the mapping vector w not have a sparse FFT, it should also be linearly independent of the input vector for optimal performance.

For large dynamic variation of data, it was found that the threshold δ under fault-free conditions does not change. This is because we scale τ to make it independent of the

magnitude of the input vector. The bitwise performance of the three EDMs deteriorates under larger dynamic data. However, we realise that often enough a bit flip at a certain bit number for larger data variation yields smaller errors in the output than the same bit flip in a smaller data variation. Thus, when we consider the only significant errors, the probability of detection does not change considerably for EDM 1 and EDM 2. For EDM 3 however, the probability of detection reduces significantly.

Experiments reveal that the probability of detection does not vary for faults injected at different stages of the FFT. In order to understand this, we studied the magnitude of errors in the output for an 8-length FFT. It was shown that the magnitude of error for faults injected in the intermediate stages depends not only on the magnitude of the error, but also the element number in the data array. The magnitude of the error in the output also depends on the magnitude of the input data element being injected with faults. Thus faults at various stages of the FFT on an average resulted in errors of similar magnitudes.

Decreasing the dynamic range of data would result in a better bit-wise coverage, although the over-all coverage for significant faults would not be effected.

8.3 Future Work

In this thesis our focus has been on faults in locations that are subsequently accessed by the application. We do not introduce any latent faults. This was because we were interested in evaluating and comparing the coverages of the three EDMs. It would be interesting to study the performance of these EDMs under more real-life-like scenarios where faults could occur at random locations in the address space. This study would reveal the susceptibility of a compute-intensive application such as this to faults in various segments of its address

space.

Our input data consisted of dynamic ranges of 10^{18} and 10^{22}. Our results show that for significant errors, the error detection capability of the EDM 1 and EDM 2 is not compromised for larger dynamic data range. We can study the performance of the EDMs under higher dynamic data ranges and verify this.

Finally, we observed that the average prob. of detection does not vary for various stages of the FFT. This was attributed largely to the variation in magnitude of data elements. It would be interesting to study the probability of detection for data with no variation or minimal variation in input data elements. This would give us a true sense of the impact of errors at various stages.

Bit	*Original Contents(X)*	*Contents after bit flip(X)*	$(X - \mathcal{X})/\mathcal{X}$
0	1.000000e+03	-1000.000000000000000	2.000000e+00
1	1.000000e+03	0.000000000000000	1.000000e+00
2	1.000000e+03	1.340781e+157	-1.340781e+154
3	1.000000e+03	1.157921e+80	-1.157921e+77
4	1.000000e+03	3.402824e+41	-3.402824e+38
5	1.000000e+03	1.844674e+22	-1.844674e+19
6	1.000000e+03	4294967296000.00000000000	-4.294967e+09
7	1.000000e+03	65536000.000000000000000	-6.553500e+04
8	1.000000e+03	3.906250000000000	9.960938e-01
9	1.000000e+03	16000.000000000000000	-1.500000e+01
10	1.000000e+03	4000.000000000000000	-3.000000e+00
11	1.000000e+03	2000.000000000000000	-1.000000e+00
12	1.000000e+03	744.000000000000000	2.560000e-01
13	1.000000e+03	872.000000000000000	1.280000e-01
14	1.000000e+03	936.000000000000000	6.400000e-02
15	1.000000e+03	968.000000000000000	3.200000e-02
16	1.000000e+03	1016.000000000000000	-1.600000e-02
17	1.000000e+03	992.000000000000000	8.000000e-03
18	1.000000e+03	1004.000000000000000	-4.000000e-03
19	1.000000e+03	1002.000000000000000	-2.000000e-03
20	1.000000e+03	1001.000000000000000	-1.000000e-03
21	1.000000e+03	1000.500000000000000	-5.000000e-04
22	1.000000e+03	1000.250000000000000	-2.500000e-04
23	1.000000e+03	1000.125000000000000	-1.250000e-04
24	1.000000e+03	1000.062500000000000	-6.250000e-05

Table 7: Effects of bit flips

A

Bit	*Original Contents(X)*	*Contents after bit flip(X)*	$(X - \mathcal{X})/\mathcal{X}$
25	1.000000e+03	1000.031250000000000	-3.125000e-05
26	1.000000e+03	1000.015625000000000	-1.562500e-05
27	1.000000e+03	1000.007812500000000	-7.812500e-06
28	1.000000e+03	1000.003906250000000	-3.906250e-06
29	1.000000e+03	1000.001953125000000	-1.953125e-06
30	1.000000e+03	1000.000976562500000	-9.765625e-07
31	1.000000e+03	1000.000488281250000	-4.882813e-07
32	1.000000e+03	1000.000244140625000	-2.441406e-07
33	1.000000e+03	1000.000122070312500	-1.220703e-07
34	1.000000e+03	1000.000061035156250	-6.103516e-08
35	1.000000e+03	1000.000030517578125	-3.051758e-08
36	1.000000e+03	1000.000015258789062	-1.525879e-08
37	1.000000e+03	1000.000007629394531	-7.629395e-09
38	1.000000e+03	1000.000003814697266	-3.814697e-09
39	1.000000e+03	1000.000001907348633	-1.907349e-09
40	1.000000e+03	1000.000000953674316	-9.536743e-10
41	1.000000e+03	1000.000000476837158	-4.768372e-10
42	1.000000e+03	1000.000000238418579	-2.384186e-10
43	1.000000e+03	1000.000000119209290	-1.192093e-10
44	1.000000e+03	1000.000000059604645	-5.960464e-11
45	1.000000e+03	1000.000000029802322	-2.980232e-11
46	1.000000e+03	1000.000000014901161	-1.490116e-11
47	1.000000e+03	1000.000000007450581	-7.450581e-12
48	1.000000e+03	1000.000000003725290	-3.725290e-12

Table 8: Effects of bit flips contd.

Bit	*Original Contents(X)*	*Contents after bit flip(X)*	$(X - \mathcal{X})/\mathcal{X}$
49	1.000000e+03	1000.000000001862645	-1.862645e-12
50	1.000000e+03	1000.000000000931323	-9.313226e-13
51	1.000000e+03	1000.000000000465661	-4.656613e-13
52	1.000000e+03	1000.000000000232831	-2.328306e-13
53	1.000000e+03	1000.000000000116415	-1.164153e-13
54	1.000000e+03	1000.000000000058208	-5.820766e-14
55	1.000000e+03	1000.000000000029104	-2.910383e-14
56	1.000000e+03	1000.000000000014552	-1.455192e-14
57	1.000000e+03	1000.000000000007276	-7.275958e-15
58	1.000000e+03	1000.000000000003638	-3.637979e-15
59	1.000000e+03	1000.000000000001819	-1.818989e-15
60	1.000000e+03	1000.000000000000909	-9.094947e-16
61	1.000000e+03	1000.000000000000455	-4.547474e-16
62	1.000000e+03	1000.000000000000227	-2.273737e-16
63	1.000000e+03	1000.000000000000114	-1.136868e-16

Table 9: Effects of bit flips contd.

REFERENCES

[1] Michael Turmon, Robert Granat, Daniel S.Katz, John Z.Lou ”*Tests and Tolerances for High-Performance Software-Implemented Fault Detection ”, IEEE transactions on Computers, Volume:52 Issue 5, May 2003*

[2] H.Wasserman and M.Blum”*Software reliability via run-time result checking”, ACM vol. 44, no.6, 1997*

[3] Andreas Steininger, Christoph Scherrer *”On Finding an Optimal Combination of Error Detection Mechanisms Based on Results of Fault Injection Experiments”,FTCS, July 1997*

[4] D.L.Tao, C.R.P Hartmann and Y.S.Chen ”*A Novel Concurrent Error Detection Scheme for FFT Networks,” Digest of Papers, The 20th International Symposium on Faul-Tolerant Computing, June 1990*

[5] F.Lombardi and J.C.Muzio ”*Concurrent Error Detection and Fault Location in an FFT Architecture”, IEEE Journal of Solid-State Circuits, May 1992, On page(s): 728-736, Volume: 27, Issue: 5*

[6] C.G.Oh and Y.Youn”*On Concurrent Error Detection, Location and Correction of FFT Networks,” Digest of Papers, The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993*

[7] C.G.Oh, Y.Youn and V.K.Raj"*An Efficient Algorithm-Based Concurrent Error Detection for FFT Networks", IEEE Transactions on Computers, Sept 1995*

[8] S.J.Wang and N.K.Jha"Algorithm Based Fault tolerance for FFT Networks", IEEE transactions on Computers, July 1995, On page(s): 849-854 Volume: 43, Issue: 7

[9] G.Robert Redinbo *Concurrent Error Detection in Fast Unitary Transform Algorithms, Dependable Systems and Networks, 2001. Proeedings. The International conference on July 2001*

[10] Mei-Chen Hsueh,Timothy K.Tsai and Ravishankar Iyer *"Fault Injection Techniques and Tools", IEEE transcations on Computer, Volume: 30 Issue: 4 , April 1997Page(s): 75 -82*

[11] J.Arlat et al. "*Fault Injection for Dependability Validation: A Methoodology and Some Applications", IEEE trans. on Software Engineering, Volume: 16 Issue: 2 , Feb. 1990Page(s): 166 -182 Feb. 1990*

[12] Z.Segall,D.Vrsalovic,et. al."*FIAT-Fault Injection Based Automated Testing Environment",18th Int. Symp. on Fault-Tolerant Computing (FTCS18) , pages 102–107, Tokyo, Japan, 1988. IEEE Computer Society Press*

[13] Ghani Kanawati,Nasser Kanawati and Jacob Abraham *"FERRARI: A Tool for the Validation of System Dependability Properties", FTCS-22: 22nd International Symposium on Fault Tolerant Computing, pages 336–344, Boston, Massachusetts, 1992. IEEE Computer Society Press.*

[14] Han,Shin,Rosenberg "*DOCTOR: An IntegrateD Software Fault InjectCTiOn Environment for Distributed Real-time Systems",IEEE Int'l Computer Performance and Dependability Symp.,1995*

[15] W.Kao,R.Iyer, and D.Tang"*FINE:A fault injection and monitoring environment for tracing the UNIX system behavior under faults",IEEE trans on Software Engineering, Volume: 19 Issue: 11 , Nov. 1993Page(s): 1105 -1118*

[16] P.Emerald Chung, Woei-Jyh Lee, Joanne Shih, Shalini Yajnik"*Fault-injection Experiments for Distributed Objects",IEEE Proceedings of the International Symposium on Distributed Objects and Applications, 1999 , 5-6 Sept. 1999Page(s): 88 -97*

[17] S.Dawson,F.Jahanian,T.Mitton, T.L.Tung "*Testing of fault-tolerant and real-time distributed systems via protocol fault-injection" Proceedings of the 26th International Symposium on Fault tolerant Computing, June 1996, pg 404-414*

[18] Timothy Tsai and RaviShankar Iyer"*FTAPE:A fault injection tool to measure fault tolerance",FTCS-26,June 1996*

[19] Aidemark, J.; Vinter, J.; Folkesson, P.; Karlsson, J."*GOOFI: generic object-oriented fault injection tool ", The International Conference on Dependable Systems and Networks, 2001 , 1-4 July 2001Page(s): 83 -88*

[20] Raphael R. Some, Won S. Kim, Khanoyan, G, L.Callum , A.Agrawal ,J.J. Bealum, Shamilian, Nikora"*Fault Injection Experiment results in Space Borne application programs" , Aerospace Conference Proceedings, 2002. IEEE , Volume: 5 , 9-16 March 2002Page(s): 5-2133 -5-2147 vol.5*

[21] Costa, D.; Rilho, T.; Vieira, M.; Madeira, H.; "*ESFFI-a novel technique for the emulation of software faults in COTS components", Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2001. , 17-20 April 2001Page(s): 197 -204*

[22] "*Algorithm-based fault tolerance for matrix operations," IEEE Trans. Computers, vol.33, no. 6, 1984*

[23] "*Algorithm Based Fault tolerance versus Result-Checking for Matrix Computations", Fault Tolerant Computing, Digest of Papers, 29th annual Internation Symposium, 1999*

[24] M.Blum and H.Wasserman "*Program Result Checking : A theory of testing meets a test of Theory," Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on , 20-22 Nov. 1994*

[25] T.Reinhart, C.Boettcher, H.Wasserman"*An automated testing methodology based on self-checking software,", Aerospace and Electronics Conference, 1998. NAECON 1998. Proceedings of the IEEE 1998 National , 13-17 July 1998*

[26] Lawrence, B."Ȧpplication of the fast Fourier number theoretic transform to radar ," Radar Conference, 1991., Proceedings of the 1991 IEEE National , 12-13 March 1991

[27] Pitas, I.; Strintzis, M.G.; "*Floating point error analysis of two-dimensional, fast Fourier transform algorithms," Circuits and Systems, IEEE Transactions on , Volume: 35 Issue: 1 , Jan. 1988*

[28] Chin-Chien Sha; Leavene, R.W. "*An algorithm-base fault tolerance (more than one error) using concurrent error detection for FFT processors ", VLSI, 1994. 'Design Automation of High Performance VLSI Systems'. GLSV '94, Proceedings., Fourth Great Lakes Symposium on , 4-5 March 1994*

[29] David S.Moore; George P.McCabe "*Introduction to the Practice of Statistics", third edition, pg.586*

[30] David S.Moore; George P.McCabe "*Introduction to the Practice of Statistics", third edition, pg.383*